“C” Programming Guide

Portable Series

Ver. 3.01

SYNTECH INFORMATION CO., LTD.

Head Office: 8F, No.210, Ta-Tung Rd., Sec.3, Hsi-Chih, Taipei Hsien, Taiwan
Tel: +886-2-2643-8866 Fax: +886-2-2643-8800
e-mail: support@cipherlab.com.tw http://www.cipherlab.com.tw

TABLE OF CONTENTS

Preface

This programming guide is meant for users to write application programs for CipherLab Portable Terminals by using the “C” Compiler with CipherLab portable specific libraries. This programming guide describes the application development process with the “C” Compiler in details. It starts with the general introduction about the features and usages of the development tools, the definition of the functions/ statements, as well as some sample programs.

Chapter 1, “Development Environment”, gives a concise introduction about the “C” Compiler and provides a step-by-step description in developing application programs for the Portable Data Terminal with the “C” Compiler.

Chapter 2, “C Compiler”, discusses some specific characteristics of the “C” Compiler.

Chapter 3, “Portable Specific Functions”, presents the user callable routines specific to the features of the Portable Data Terminals.

Chapter 4, “Standard Library Routines”, the standard ANSI library routines are briefly described, as the more detailed information can be found in many ANSI C related literature.

Chapter 5, “Real Time Kernel”, discusses the concepts of the real time kernel, μC/OS. Users can generate a real time multitasking system by using the μC/OS functions.

1 Development Environment

1.1 Directory Structure

The CipherLab Portable Terminals "C" Language Development Kit contains six directories, namely, **BIN**, **ETC**, **INCLUDE**, **LIB**, **README**, and **USER**. The purposes/contents of each directory are listed below.

1) BIN : This directory contains 18 files.
   - 16 execution files for compilation, linking and so on,

     asm900.exe, cc900.exe, dos4gw.exe, f_amd4.exe,
     mac900.exe, pminfo.exe, privatxm.exe, rminfo.exe,
     thc1.exe, thc2.exe, tuapp.exe, tuconv.exe,
     tufal.exe, tulib.exe, tulink.exe, tumpl.exe
   - Download.exe : for downloading program via RS-232 port

   Usage of these executable files will be described further in later sections.

2) ETC : 11 files, help and version information of the "C" compiler

3) INCLUDE
   - 15 "C" header files for standard library routines

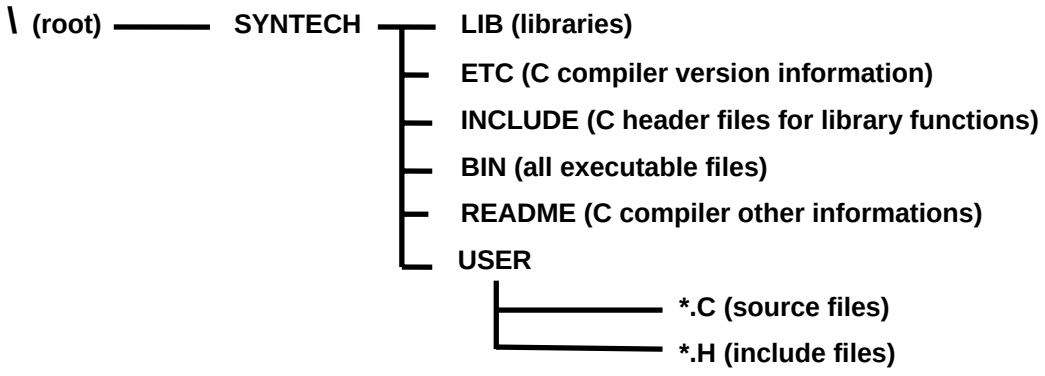
     assert.h ctype.h errno.h float.h limits.h
     locale.h math.h setjmp.h signal.h stdarg.h
     stddef.h stdio.h stdlib.h string.h time.h
   - 1 header file for terminal specific library : e.g. 711lib.h
   - 1 header file for Real Time Kernel Library : ucos.h

4) LIB : Library object code files
   - "C" standard library : c900ml.lib
   - Portable specific library : 711lib.lib, 720lib.lib, 8000lib.lib, 8100lib.lib, 8300lib.lib

5) README : C compiler version update and supplemental information

6) USER : contains the source code of the user's program or other sample programs.

To set up the "C" language development environment for the portable terminals, you can create the **\SYNTECH** directory from the root directory and then copy the above six directories to the **\SYNTECH** directory as follows:



1.2 Setup

Before using the compiler's software programs, some environmental variables must be added to the autoexec.bat.

1) path = (your own path);c:\SYNTECH\BIN

   So all executable files (.EXE & .BAT) can be found.

2) set THOME900=c:\SYNTECH

   This is a must for the C compiler to locate all necessary files

3) set tmp = c:\tmp

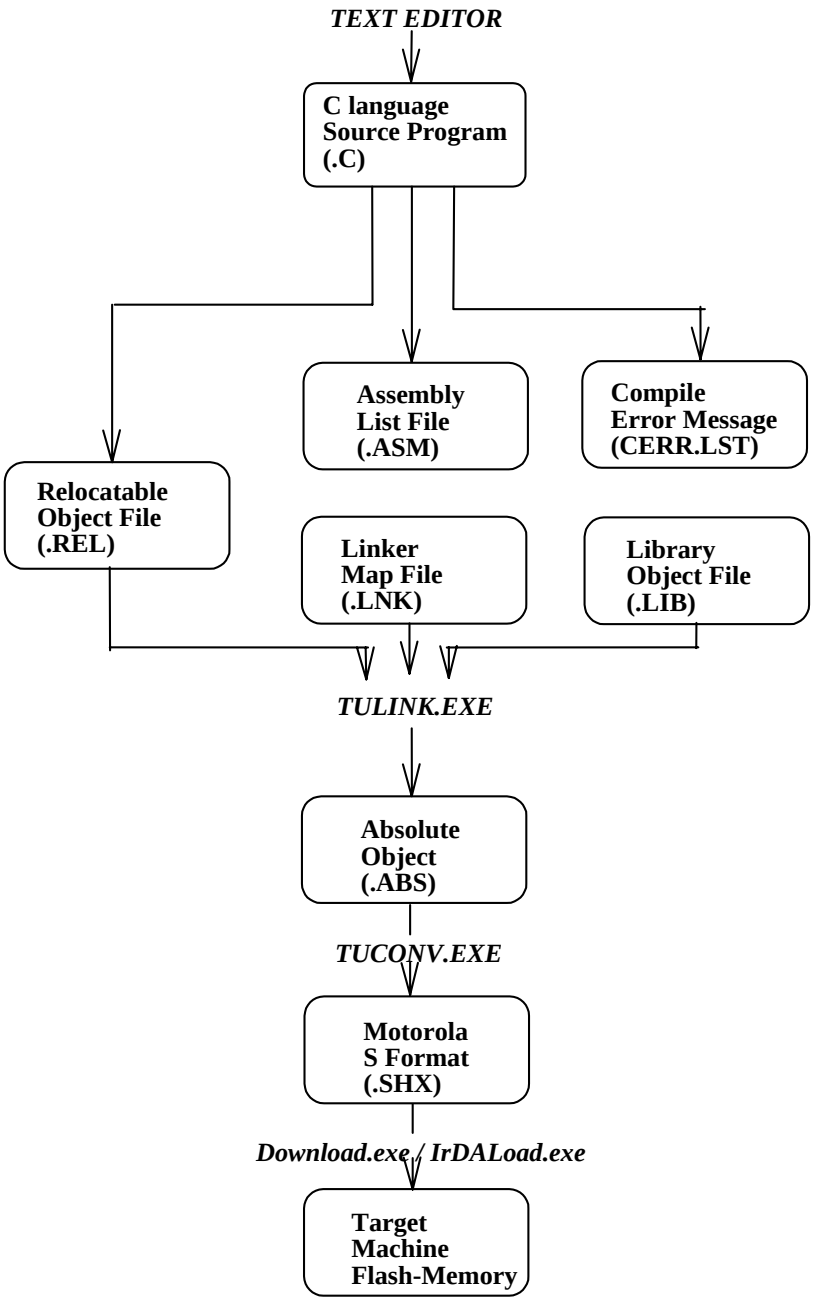
   Skip this if tmp is already specified.

Step 3 can be ignored if tmp was already specified. This is the temporary working directory for the compiler and linker (for memory and file swapping).

To improve efficiency, the compiler invokes a virtual memory manager "DOS4GW". It recognizes and supports various PCs. However, if it does not work on your PC, the program PMINFO can be used to identify the problem. (If you have problems in using the compiler, run the PMINFO, print all messages and then contact Syntech)

1.3 Development Flow

The development process is much like writing any other “C” programs on PC. The flow is illustrated as below,



1.3.1 Create Your Own “C” source program

The first step is to create or modify the desired “C” programs using any text editors. It is recommended to use ".C" as the file extension and create them under the USER directory, and use the "USER" directory as the working directory. It is also recommended to separate the whole programs into modules while retaining function integrity, and put modules into separate files to reduce compilation time.

1.3.2 Compile

To compile the “C” programs, use *cc900* command in the directory of the target file.

CC900 –[options] *FILENAME.C*

For the usage of the *cc900* command and the options, please refer to the cc900.hlp in the ETC subdirectory.

The batch file “**Y.BAT**” which can be found under the *USER* directory has been created to simplify the compiling process.

Y *FILENAME.C*

This batch file invokes the “C” compilation program which in turn calls many other executable programs under the *BIN* directory. As these programs are invoked by the compiler sequentially, their usages can be ignored. Also, many parameters are set in calling the compiler driver to accommodate target machine environments. It is recommended to use the Y.BAT file directly. If you attempt write your own batch file, remember to put the same parameters. These parameters are listed below,

- -XA1, -XC1, -XD1, -Xp1 : alignment setting, all 1
- -XF : no deletion of assembly file, if examining the assembly file is not necessary, this option can be removed
- -O3 : set optimization level (can be 0 to 3, no to maximum optimization). If code size and performance is not a problem, this option can be removed which will then set to the default -O0, that is, no optimization at all. If optimization is enabled, care must be taken that some instructions might be optimized and removed. For example,

```
test()
{
   unsigned int old_msec;
   old_msec=sys_msec;
   while (old_msec == sys_msec) ;
}
```

This routine waits till sys_msec changed. And sys_msec is a system variable that is updated each 5 ms by background interrupt. If optimization is enabled, this whole routine is truncated as it is meaningless (which is a dead-loop). To avoid this, the type qualifier "**volatile**" can be used to suppress optimization.

- -c : create object but no link
- -e cerr.lst : create error list file "cerr.lst"

After compilation is completed, a relocatable object file named "*program_name*.rel" is created which can be used later by the linker to create the executable object program. As the compiler compiles the program into assembler form during the process, an accompanying assembler source file "*program_name*.asm" is also created. This file helps in debugging if necessary. If any error occurs, they will be put into the file "CERR.LST" for further examination.

1.3.3 Link

If the C source programs are successfully compiled into relocatable object files. The linker must be used to create the absolute objects and then the file can be downloaded to the target machine’s flash memory for execution. However, a linker map file must be created,.

TULINK *FILENAME.LNK*

This map file "FILENAME.LNK" is used to instruct the linker to allocate absolute addresses of code, data, constant and so on according to the target machine environments. This is a lengthy process as it depends on the hardware architecture. Fortunately, a sample linker map file is provided and few steps are required to customize it for your own need, while leaving hardware-related stuff unchanged.

As you can see from the sample linker file listed as follows, the only parts have to be changed is the file names (under-lined & bolded sections). If linked successfully, an absolute object file named "FILE1.ABS" is created. Also a file named "FILE1.MAP" lists all code, variable addresses and error messages if any.

```
-lm -lg             /* parameters for TULINK, don't change */
FILE1.REL           /* your C program name */
FILE2.REL           /* your C program name */
....
....
FILEN.REL           /* your C program name */
..\lib\c900ml.lib   /* standard library */
..\lib\720lib.lib   /* 720 Function library */

/*****************************************/
/*  User could provide suitable values to  */
/*    the following two variables          */
/*****************************************/
MainStackSize = 0x001000;
HeapSize  = 0x000100;

/*****************************************/
/*  Do not modify anything beyond this line  */
/*****************************************/
memory
{
    IRAM: org = 0x001100, len = 0x000e00    /* 0x1000 - 0x10ff IntVec */
                                            /* 0x1f00 - 0x1fff Stack */
    RAM      : org = 0x804000, len = 0x01c000
    ROM      : org = 0xf00000, len = 0x0e0000
}

sections
{
    code  org = 0xf00000 : {
        *(f_head)
        *(f_code)
    } > ROM

    area org = 0x804000 : {
        . += MainStackSize;
        . += HeapSize;
        *(f_bcr)
        *(f_area)
    } > RAM

    data  org=org(code)+sizeof(code)  addr=org(area)+sizeof(area) : {
        *(f_data)
    } /* global variables with initial values */

    xcode org = org(data) + sizeof(data)  addr = addr(data) + sizeof(data) : {
        *(f_xcode)      /* code reside on RAM */
    }
    const org = org(xcode) + sizeof(xcode) : {
        *(f_const)
        *(f_tail)
    } > ROM
}

SysRamEnd      = addr(xcode) + sizeof(xcode);
DataRam        = addr(data);
CodeRam        = addr(xcode);
HeapTop        = org(area) + MainStackSize;

/* End */
```

1.3.4 Format Convertion

The absolute object file created by TULINK is in TOSHIBA's own format, before downloading it to the target terminal, it must be converted to the Motorola S format by using the “TUCONV” utility.

TUCONV -Fs32 -o *FILENAME*.shx *FILENAME*.abs

The file extension ".shx" is a must for the code downloader.

The batch file “**Z.BAT**” which can be found under the USER directory has been created to simplify the linking and format convertion process. Just run the batch file as follow:

Z

The target executable file (with SHX extension) will then be generated if no error found.

1.3.5 Download Program to Flash Memory

Now if the Motorola S format object file *FILENAME*.shx is created successfully, it can be downloaded to the flash memory for testing. Please run the DOWNLOAD.EXE utility (or IRLOAD.EXE for IR interface) and configure the following parameters properly:

- **FILENAME** *:* the file name of the absolute object code.
- **COMPORT** : select the appropriate COM port for transmission.
- **BAUDRATE** : supported baud rates are 115200, 76800, 57600, 38400, 19200, 9600, 4800, 2400.
- **PARITY** : should be no parity.
- **DATABITS** : 8

The selected baud rate, parity and data bits must match the target machine’s COM port settings.

1.4 C Compiler

This C compiler is for TOSHIBA TLCS-900 family 16-bit MCUs. It is mostly ANSI compatible. However, some specific characteristics are listed below,

1.4.1 Size of Types

Type	Size in byte
char, unsigned char	1
short int, unsigned short int, int, unsigned int	2
long int, unsigned long int,	4
pointer	4
structure, union	4

1.4.2 Representation Range of Integers

Macros concerning the representation ranges of the values of integer types are defined in the header file <limits.h> as below,

Macro Name	Contents
CHAR_BIT	number of bits in a byte (the smallest object)
SCHAR_MIN	minimum value of signed char type
SCHAR_MAX	maximum value of signed char type
CHAR_MIN	minimum value of char type
CHAR_MAX	maximum value of char type
UCHAR_MAX	maximum value of unsigned char type
MB_LEN_MAX	number of bytes in a wide character constant
SHRT_MIN	minimum value of short int type
SHRT_MAX	maximum value of short int type
USHRT_MAX	maximum value of unsigned short int type
INT_MIN	minimum value of int type
INT_MAX	maximum value of int type
UINT_MAX	maximum value of unsigned int type
LONG_MIN	minimum value of long int type
LONG_MAX	maximum value of long int type
ULONG_MAX	maximum value of unsigned long int type

1.4.3 Floating Types

Float types are supported and conforms to IEEE standards,

Type	Size in bits
float	32
double	64
long double	64

1.4.4 Alignment

Alignments of different types can be adjusted. This is to facilitate CPU performance while sacrificing memory spaces. However as all target systems utilize 8-bit data bus, the alignment does not effect performance and is fixed to 1 for all types. In invoking the C compiler driver -XA1, -XD1, -XC1 and -Xp1 is specified.

1.4.5 Register and Interrupt Handling

These are possible through C. However, they are inhibited as all accessing to system resources should be made via Syntech library routines.

1.4.6 Reserved Words

Basic reserved (common to all Cs) words are listed below,

auto	double	int	struct	break
else	long	switch	case	enum
register	typedef	char	extern	return
union	const	float	short	unsigned
continue	for	signed	void	default
goto	sizeof	volatile	do	if
static	while			

1.4.7 Extended Reserved Words

These reserved words are specific to this C compiler and all of them start with "_ _", two underscores.

_ _adcel	_ _cdcel	_ _near	_ _far
_ _tiny	_ _asm	_ _io	
_ _XWA	_ _XBC	_ _XDE	_ _XHL
_ _XIX	_ _XIY	_ _XIZ	_ _XSP
_ _WA	_ _BC	_ _DE	_ _HL
_ _IX	_ _IY	_ _IZ	_ _W
_ _A	_ _B	_ _C	_ _D
_ _E	_ _H	_ _L	_ _SF
_ _ZF	_ _VF	_ _CF	
_ _DMAS0	_ _DMAS1	_ _DMAS2	_ _DMAS3
_ _DMAD0	_ _DMAD1	_ _DMAD2	_ _DMAD3
_ _DMAC0	_ _DMAC1	_ _DMAC2	_ _DMAC3
_ _DMAM0	_ _DMAM1	_ _DMAM2	_ _DMAM3
_ _NSP	_ _XNSP	_ _INTNEST	

1.4.8 Bit-Field Usage

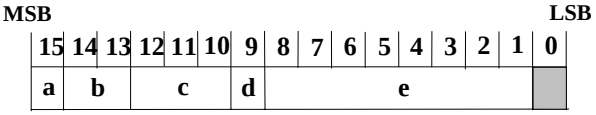
The following types can be used as the bit field base types.

Type	Bits
char, unsigned char	8
short int, int, unsigned short int, unsigned int	16
long int, unsigned long int	32

The allocation is made as follows,

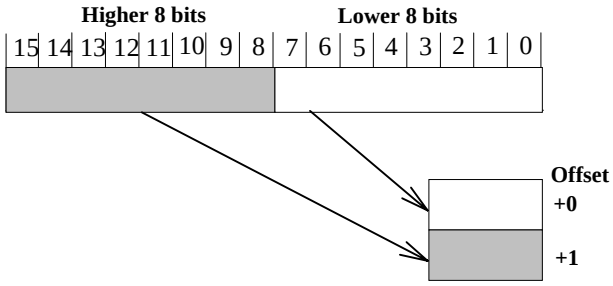
1) Fields are stored from the highest bits

```
struct field1 {
        unsigned        int     a:1;
        unsigned        int     b:2;
        unsigned        int     c:3;
        unsigned        int     d:1;
        unsigned        int     e:8;
}
```



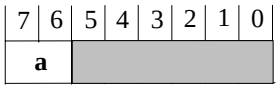
2) Little endien

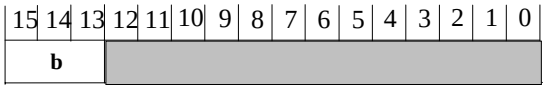
If the base type of a bit field member is a type requiring two bytes or more (e.g. unsigned int), the data is stored in memory after its bytes are turned topside down.



3) Different types : A bit field with different type is assigned to a new area

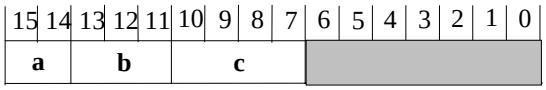
```
struct field {
        unsigned   char       a:2;
        unsigned    short     b:3;
}
```





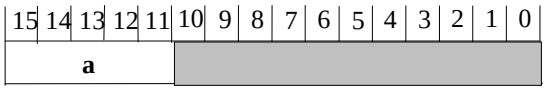
4) Different type (signed/unsigned)

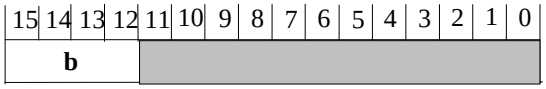
```
struct field {
        signed          short    a:2;
        unsigned short           b:3;
        signed          short    c:4;
}
```



5) Different type (same size)

```
struct field {
        signed          short    a:5;
        unsigned int             b:4;
}
```





The bit-field can be very useful in some cases. However, if memory is not a concern, it is recommended not to use the bit-fields. As the code size and performance are degraded.

2 Terminal Specific Function Library

There are several terminal specific library routines to facilitate the development of the user's application. These functions cover a wide variety of tasks, including communications, show string or bitmap on LCD, buzzer control, scanning, file manipulation, etc. They are categorized and described in this chapter by their functions or the resources they work on. The function prototypes of the library routines and the declaration of the system variables can be found in the library header file, for example, "8300lib.h". It is assumed that the programmer has prior knowledge of "C" language.

2.1 System

2.1.1 Power On Reset (POR)

After reset, a portion of library functions called POR routine initializes the system hardware, memory buffers, and parameters such as follows,

- RS232 : all disabled
- reader ports : all disabled
- keypad scanning : enabled
- LCD display : initialised and cleared to blank, cursor is on and set to the upper-left corner (0,0)
- calendar chip : initialised
- LEDs : all off
- Allocate stack area and other parameters

There must be one and only one "**main**" function in the C program which is the entry point of the application program. Control is then transferred to the "main" function whenever the system initialisation is done.

2.1.2 System Global Variables

There are several global variables that are declared by the system, two of them are system timers that are cleared to 0 upon power up. As they are updated by the timer interrupt, please do NOT write to them.

- extern volatile unsigned long sys_msec; /* in unit of 5 ms */
- extern volatile unsigned long sys_sec; /* in unit of 1 second */

Other system variables are as follows,

- extern unsigned int AUTO_OFF;

  This variable governs the time for the system to automatically shut down the user's program whenever there is no operation during the preset period. The unit for this variable is second, and if it's set to zero, the AUTO_OFF function will be disabled.

- extern int BC_X, BC_Y; /* for 8000 and 8300 only */

  These two variables govern the location of the battery icon, change their values will change the battery icon's location. For 8000, their default values are 96 and 51. For 8300, they are 120 and 51.

- extern int IrDA_Timeout;

  This variable governs the timeout for the IrDA connection, i.e., the system will give up trying to establish a connection with an IrDA device after trying out this time period. The valid settings for this variable are ranging from 1 to 8, which represent the following time periods:

1	3 sec
2	8 sec
3	12 sec
4	16 sec
5	20 sec

6 25 sec
7 30 sec
8 40 sec

The default value for this setting is 1, i.e., 3 seconds.

- extern int KEY_CLICK [4]; /* for 8000 and 8300 only */

  This variable holds the sound frequency / duration pair of the key click. The following example can be used to generate a beep same as the key click.

  on_beeper (KEY_CLICK);

- extern unsigned int POWER_ON;

  This variable can be set to either POWERON_RESUME or POWERON_RESTART. The default is POWERON_RESUME, i.e., upon power up, the user program will start from the status of last power off. Note that if the user removes the batteries and then reloads batteries, or by entering system menu before normal operation, the user program will always restart itself upon power up.

- char ProgVersion [16];

  This characters array can be used to store the version information of the user's program. This version information can be checked from the Version submenu of the system menu. Note your “C” program needs to declare this variable to overwrite the system default setting. For example,

  char ProgVersion [16] = "Power AP 1.00";

- unsigned char WakeUp_Event_Mask; /* for 8300 only */

  For 8300 series, it’s possible to wake up the terminal by the following predefined events:

  Wedge_WakeUp : wake up once the keyboard-wedge cable is connected
  RS232_WakeUp : wake up once the RS-232 cable is connected
  Charging_WakeUp : wake up once the terminal is being charged
  ChargeDone_WakeUp : wake up once the battery charging is done

  For example,

  WakeUp_Event_Mask = RS232_WakeUp | Charging_WakeUp;

  /* wake up by RS-232 cable connection or battery charging events. */

ChangeSpeed

purpose	To change the CPU’s running speed
syntax	void ChangeSpeed (int speed);
example	ChangeSpeed (4);
description	If high-speed operation is not required, selecting low CPU speed can save battery power. There are five speeds available: 1, 2, 3, 4, and 5, which represent sixteenth, eighth, quarter, half and full speed of the CPU respectively.
return	none

DownloadPage

purpose	Stop the application and force the system to jump to the system menu for downloading new programs.
syntax	void DownloadPage (void);
example	DownloadPage ();

description For 8000 and 8300 series, it is possible to pass arguments to suppress the download menu. For example,

DownloadPage (NO_MENU, COMM_DIRECT, BAUD_115200);

The first parameter must be the NO_MENU constant. The 2nd parameter is the communication type. The 3rd parameter is the transmission baud rate. For contants of the 2nd and 3rd parameters, please check the header files. In these cases, the terminal will be set to the "Ready to download" state without prompting the download menu.

return none

_KeepAlive__

purpose To keep user's application program continuous running without automatic shutting down by the system.

syntax void _KeepAlive__(void);

example _KeepAlive__();

description Whenever this routine is called, it will reset the counter governed by the global variable AUTO_OFF so that user's application program will keep on running without automatic shutting down by the system.

return none

shut_down

purpose Shut down the system.

syntax void shut_down (void);

example shut_down();

description This routine will shut down the system. Upon power up, the system will always restart.

return none

SysSuspend

purpose Shut down the system.

syntax void SysSuspend (void);

example SysSuspend();

description This routine will shut down the system. Upon power up, the system will resume or restart itself, depending on the system setting.

return none

system_restart

purpose Re-start the system

syntax void system_restart (void);

example system_restart();

description The routine jumps to the power on reset point and restarts the system.

return none

2.1.3 Security

The system menu can be password-protected. To protect the user from entering the system menu without an authority, you can either directly enable the password-protected mechanism from within the system menu or through programming. Besides, there are several security related functions available so that you can use the same password to protect your own application.

CheckPasswordActive

purpose	To check if the system password is enabled or not.
syntax	int ChechPasswordActive (void);
example	if (ChechPasswordActive()) printf (“Please input password:”);
description	This function detects if the system password is enabled or not. By default, the system menu is not password-protected.
return	0 if it’s disabled, 1 if enabled.
See also	CheckSysPassword, InputPassword, SaveSysPassword

CheckSysPassword

purpose	To check if the input string matches the system password.
syntax	int ChechSysPassword (const char *psw);
example	if (!ChechSysPassword (szInput)) printf (“Password incorrect !!!”);
description	If the system password is enabled and you want to use the same password to protect your application, then this function can be used to check if the input string matches the system password.
return	1 represents match, 0 represents mismatch.
See also	CheckPasswordActive, InputPassword, SaveSysPassword

InputPassword

purpose	To let the user input the password.
syntax	int InputPassword (char *psw);

example	char szPsw [10];
	printf (“Input password:”);
	if (InputPassword (szPsw))
	  if (!ChechSysPassword (szPsw))
	  printf (“Illegal password!”);

description	This function provides a simple edit control for the user to input the password, but instead of showing normal characters, it shows an asteroid (*) on the display whenever the user inputs a character.
return	1 if the user presses enter key to confirm the input, 0 if the user presses ESC to cancel the input.
See also	CheckPasswordActive, CheckSysPassword, SaveSysPassword

SaveSysPassword

purpose	To save / change the system password.
syntax	int SaveSysPassword (const char *psw);
example	SaveSysPassword (“12345”);
description	This function allows you to change the system password, but the length of the password can not longer than 8 characters. If the input string is null, the system password will be disabled.
return	1 if successful, 0 if the length of the password is longer than 8 characters.
See also	CheckPasswordActive, CheckSysPassword, InputPassword

2.1.4 Program Manager

The Program Manager is part of the kernel for 8000 and 8300 series. The 1MB flash memory is divided into 16 banks, which is 64KB each. The kernel itself takes 2 banks, and the system reserves one bank for data storage. If the user does not download a font file to the system, then there are still 13 banks available for storing user programs. It is possible to store up to 6 programs, but only one of them can be activated and then will be running upon power up. For operation of the Program Manager, please refer to 8000 and 8300 User’s Manual. The system provides the following functions for managing the multiple programs directly.

ActivateProgram

purpose	To make one of the resident programs become the active program.
syntax	void ActivateProgram (int Prog); int Prog; /* 1 ~ 6, represents one of the 6 resident programs */
example	ActivateProgram (3); /* make the 3rd program become active */
description	This function copies the designated program to the active area and make it become the active program. The original program resided on the active area will be then replaced by the new program.
return	none.
See also	LoadProgram, ProgramManager, ProgramInfo, UsedBank

LoadProgram

purpose	To load a user program (*.SHX) to the designated location.
syntax	void LoadProgram (int Prog); int Prog; /* 1 ~ 6, represents one of the 6 resident locations */
example	LoadProgram (3); /* Load the user program to the 3rd location */
description	Upon calling this function, the system jumps to the download page for downloading the user program to the designated location.
return	none.
See also	ActivateProgram, ProgramManager, ProgramInfo, UsedBank

ProgramInfo

purpose	To check the name and size of the designated program.
syntax	unsigned int ProgramInfo (int Prog, char *NameStr); int Prog; /* 1 ~ 6, represents one of the 6 resident locations */

char *NameStr; /* char pointer for receiving the program name */

example unsigned char ProgName [20];
int nSize = ProgramInfo (1, ProgName); /* get info of the 1st program */

description This function is used to retrieve program information including its size and name. The program name is the one that shown inside the menu of Program Manager.

return The size of memory banks in kilo-bytes that occupied by the program. Since one bank is 64KB, the return value will be 64, 128, …, etc.

See also ActivateProgram, LoadProgram, ProgramManager, UsedBank

ProgramManager

purpose To enter the kernel and bring up the menu of the Program Manager.

syntax void ProgramManager (void);

example ProgramManager (); /* Jump to the menu of the Program Manager */

description Upon calling this function, the program will stop running and jump to the kernel and then the Program Manager will take over the control.

return none.

See also ActivateProgram, LoadProgram, ProgramInfo, UsedBank

UsedBank

purpose To check how many flash banks are used by user programs.

syntax unsigned int UsedBank (void);

example unsigned int nUsed = UsedBank(); /* check flash memory */

description This function is used to check the availability of the flash memory.

return The return value is a 16-bit unsigned integer, which each bit represents the status of one memory bank. If the bit value is 0, then the related memory bank is occupied, otherwise (bit value is 1) it's still available for storing program.

See also ActivateProgram, LoadProgram, ProgramInfo, ProgramManager

2.2 Reader

The barcode decoding routines consist of 3 functions: **InitScanner1( )**, **Decode( )**, and **HaltScanner1( )**. The *InitScanner1( )* is used to initialise the scanner port. The *Decode( )* function is used to perform decoding. And the *HaltScanner1( )* is used to stop the scanner port from operating.

2.2.1 Barcode and Magnetic Card Decoding

To enable barcode decoding capability in the system, the scanner port must be first initialised by calling the *InitScanner1( )* function. After the scanner ports is initialised, the *Decode( )* function can be called in the program loops to perform barcode decoding.

There are four global variables relate to the barcode decoding routines: **ScannerDesTbl**, **CodeBuf**, **CodeLen**, and **CodeType**. These variables are declared by the system, the user program needs not to declare them.

ScannerDesTbl : This 23 bytes of unsigned character array governs the operation of the *Decode* routine.

CodeBuf : This buffer contains the decoded data upon successful decoding.

CodeLen : This integer indicates the length of the decoded data upon successful decoding.

CodeType : This character indicates the type of code (symbology) being decoded upon successful decoding.

2.2.2 Code Type

The following list shows the possible values of the *CodeType* variable.

Name	Type	Name	Type
Code 39	A	UPCE with Addon 2	K
Italy Pharma-code	B	UPCE with Addon 5	L
CIP 39	C	EAN8 no Addon	M
Industrial 25	D	EAN8 with Addon 2	N
Interleave 25	E	EAN8 with Addon 5	O
Matrix 25	F	EAN13 no Addon	P
Codabar (NW7)	G	EAN13 with Addon 2	Q
Code 93	H	EAN13 with Addon 5	R
Code128	I	MSI	S
UPCE no Addon	J	Plessey	T

2.2.3 Scanner Description Table

The unsigned character array ***ScannerDesTbl*** governs the behavior of the Decode function. The following table describes the details of the ***ScannerDesTbl*** variable.

Subscriptor	Bit	Description
0	7	1 : Enable Code 39 0 : Disable Code 39
0	6	1 : Enable Italy Pharma-code 0 : Disable Italy Pharma-code
0	5	1 : Enable CIP 39 0 : Disable CIP 39
0	4	1 : Enable Industrial 25 0 : Disable Industrial 25
0	3	1 : Enable Interleave 25 0 : Disable Interleave 25
0	2	1 : Enable Matrix 25 0 : Disable Matrix 25
0	1	1 : Enable Codabar (NW7) 0 : Disable Codabar (NW7)
0	0	1 : Enable Code 93 0 : Disable Code 93
1	7	1 : Enable Code 128 0 : Disable Code 128
1	6	1 : Enable UPCE no Addon 0 : Disable UPCE no Addon
1	5	1 : Enable UPCE Addon 2 0 : Disable UPCE Addon 2
1	4	1 : Enable UPCE Addon 5 0 : Disable UPCE Addon 5
1	3	1 : Enable EAN8 no Addon 0 : Disable EAN8 no Addon
1	2	1 : Enable EAN8 Addon 2 0 : Disable EAN8 Addon 2
1	1	1 : Enable EAN8 Addon 5 0 : Disable EAN8 Addon 5
1	0	1 : Enable EAN13 no Addon 0 : Disable EAN13 no Addon
2	7	1 : Enable EAN13 Addon 2 0 : Disable EAN13 Addon 2
2	6	1 : Enable EAN13 Addon 5 0 : Disable EAN13 Addon 5
2	5	1 : Enable MSI 0 : Disable MSI
2	4	1 : Enable Plessey 0 : Disable Plessey
2	3	Reserved
2	2 – 0	Reserved
3	7 – 0	Reserved
4	7 – 0	Reserved

continued on next page

continued from previous page

Subscriptor	Bit	Description
5	7	1 : Transmitting Code 39 Start/Stop Character 0 : No Transmitting Code 39 Start/Stop Character
5	6	1 : Verifying Code 39 Check Character 0 : No Verifying Code 39 Check Character
5	5	1 : Transmitting Code 39 Check Character 0 : No Transmitting Code 39 Check Character
5	4	1 : Full ASCII Code 39 0 : Standard Code 39
5	3	1 : Transmitting Italy Pharmacode Check Character 0 : No Transmitting Italy Pharmacode Check Character
5	2	1 : Transmitting CIP39 Check Character 0 : No Transmitting CIP39 Check Character
5	1	1 : Verifying Interleave 25 Check Digit 0 : No Verifying Interleave 25 Check Digit
5	0	1 : Transmitting Interleave 25 Check Digit 0 : No Transmitting Interleave 25 Check Digit
6	7	1 : Verifying Industrial 25 Check Digit 0 : No Verifying Industrial 25 Check Digit
6	6	1 : Transmitting Industrial 25 Check Digit 0 : No Transmitting Industrial 25 Check Digit
6	5	1 : Verifying Matrix 25 Check Digit 0 : No Verifying Matrix 25 Check Digit
6	4	1 : Transmitting Matrix 25 Check Digit 0 : No Transmitting Matrix 25 Check Digit
6	3 - 2	Select Interleave25 Start/Stop Pattern 00 : Use Industrial25 Start/Stop Pattern 01 : Use Interleave25 Start/Stop Pattern 10 : Use Matrix25 Start/Stop Pattern 11 : Undefined
6	1 – 0	Select Industrial25 Start/Stop Pattern 00 : Use Industrial25 Start/Stop Pattern 01 : Use Interleave25 Start/Stop Pattern 10 : Use Matrix25 Start/Stop Pattern 11 : Undefined
7	7 – 6	Select Matrix25 Start/Stop Pattern 00 : Use Industrial25 Start/Stop Pattern 01 : Use Interleave25 Start/Stop Pattern 10 : Use Matrix25 Start/Stop Pattern 11 : Undefined
7	5 – 4	Codabar Start/Stop Character 00 : abcd/abcd 01 : abcd/tn*e 10 : ABCD/ABCD 11 : ABCD/TN*E
7	3	1 : Transmitting Codabar Start/Stop Character 0 : No Transmitting Codabar Start/Stop Character
7	2 – 0	Reserved
8	7 – 0	Reserved

continued on next page

continued from previous page

Subscriptor	Bit	Description
9	7 - 6	MSI Check Digit Verification 00 : Single Modulo 10 01 : Double Modulo 10 10 : Modulo 11 and Modulo 10 11 : Undefined
9	5 - 4	MSI Check Digit Transmission 00 : the last Check Digit is not transmitted 01 : both Check Digits are transmitted 10 : both Check Digits are not transmitted
9	3	1 : Transmitting Plessey Check Characters 0 : No Transmitting Plessey Check Characters
9	2	1 : Converting Standard Plessey to UK Plessey 0 : No Converting
9	1	1 : Converting UPCE to UPCA 0 : No Converting
9	0	1 : Converting UPCA to EAN13 0 : No Converting
10	7	1 : Enable ISBN Conversion 0 : No Conversion
10	6	1 : Enable ISSN Conversion 0 : No Conversion
10	5	1 : Transmitting UPCE Check Digit 0 : No Transmitting UPCE Check Digit
10	4	1 : Transmitting UPCA Check Digit 0 : No Transmitting UPCA Check Digit
10	3	1 : Transmitting EAN8 Check Digit 0 : No Transmitting EAN8 Check Digit
10	2	1 : Transmitting EAN13 Check Digit 0 : No Transmitting EAN13 Check Digit
10	1	1 : Transmitting UPCE System Number 0 : No Transmitting UPCE System Number
10	0	1 : Transmitting UPCA System Number 0 : No Transmitting UPCA System Number
11	7	1 : Converting EAN8 to EAN13 0 : No Converting
11	6	Reserved
11	5	Reserved
11	4	1 : Enable Negative Barcode 0 : Disable Negative Barcode
11	3 – 2	00 : No Read Redundancy for Scanner Port 1 01 : One Time Read Redundancy for Scanner Port 1 10 : Two Times Read Redundancy for Scanner Port 1 11 : Three Times Read Redundancy for Scanner Port 1
11	1 – 0	Reserved

continued on next page

continued from previous page

Subscriptor	Bit	Description
12	7	1 : Industrial 25 Code Length Limitation in Max/Min Length Format 0 : Industrial 25 Code Length Limitation in Fix Length Format
12	6 – 0	Industrial 25 Max Code Length / Fixed Length 1
13	7 – 0	Industrial 25 Min Code Length / Fixed Length 2
14	7	1 : Interleave 25 Code Length Limitation in Max/Min Length Format 0 : Interleave 25 Code Length Limitation in Fix Length Format
14	6 – 0	Interleave 25 Max Code Length / Fixed Length 1
15	7 - 0	Interleave 25 Min Code Length / Fixed Length 2
16	7	1 : Matrix 25 Code Length Limitation in Max/Min Length Format 0 : Matrix 25 Code Length Limitation in Fix Length Format
16	6 - 0	Matrix 25 Max Code Length / Fixed Length 1
17	7 - 0	Matrix 25 Min Code Length / Fixed Length 2
18	7	1 : MSI Code Length Limitation in Max/Min Length Format 0 : MSI Code Length Limitation in Fix Length Format
18	6 - 0	MSI 25 Max Code Length / Fixed Length 1
19	7 - 0	MSI Min Code Length / Fixed Length 2
20	7 - 4	Scan Mode for Scanner Port 1 0000 : Auto Off Mode 0001 : Continuous Mode 0010 : Auto Power Off Mode 0011 : Alternate Mode 0100 : Momentary Mode 0101 : Repeat Mode 0110 : Laser Mode 0111 : Test Mode 1000 : Aiming Mode
20	3 - 0	Reserved
21		Scanner Time-out Duration in seconds for Auto Off and Auto Power Off scanning modes.
22		Reserved

2.2.4 Scan Modes

The scanner supports up to 9 scanning modes as described below.

- **Auto Off Mode :** The scanner will start scanning once the switch is triggered. The scanning continues until either a barcode is read or preset scanning period (scanner Time-Out duration) is expired.
- **Continuous Mode :** The scanner is always scanning but just decode once for the same barcode.
- **Auto Power Off Mode :** The scanner will start scanning once the switch is triggered. The scanning continues until a preset scanning period (scanner Time-Out duration) is expired. Unlike the Auto-Off mode, the scanner will continue to scan and the scanning period is re-counted each time there is a successful read.
- **Alternate Mode :** The scanner will start scanning once the switch is triggered. The scanner will keep on scanning until the switch is triggered again.
- **Momentary Mode :** The scanner will be scanning as long as the switch is depressed.
- **Repeat Mode :** The scanner is always scanning just like the Continuous Mode. But now the switch acts like a "re-transmit button". If the switch is triggered within 1 second after a good read, the same data will be transmitted again without actually reading the barcode. The "re-transmit button" can be triggered as many times as user needs, so long as the time between each trigger does not exceed 1 second. This scan mode is very useful when the same barcode is to be read many times.
- **Laser Mode :** This is the scan mode used most often on laser scanners. The scanner will start scanning once the switch is pressed. The scanning goes on until either a barcode is read or the switch is released.
- **Test Mode :** The scanner is always scanning and will decode repeatedly even with the same barcode.
- **Aiming Mode :** By selecting this mode, user needs to trigger twice for a decoding. That is, the first trigger is for aiming only, and the second trigger will trully start to decode. After first trigger, the scanner will keep on scanning for one second so that user may take aim. But user must press the second trigger within this period (default to one second), otherwise it will be reset and user has to take aim again. This mode is used when two consecutive barcodes are printed too closed that users need to take aim and make sure they don't read the wrong barcode. There is a system global variable *AIMING_TIMEOUT* that can be used to change the default one-second time-out duration. The unit for this variable is 5ms.

Decode

purpose	Perform barcode decoding.
syntax	int Decode (void);
example call	while (1) { if (Decode( )) break; }
description	Once the scanner port is initialized (by use of *InitScanner1* function), call this *Decode* function to perform barcode decoding. This function should be called constantly in user's program loops when barcode decoding is required. If the barcode decoding is not required for a long period of time, it is recommended that the scanner port should be stopped by use of the *HaltScanner1* function. If the *Decode* function decodes successfully, the decoded data will be placed in the string variable *CodeBuf* with a string terminating character appended. And the integer variable *CodeLen*, and the character variable *CodeType* will reflect the length and the code type of the decoded data respectively.
return	Upon successful decoding, the *Decode* function returns an integer whose value equals to the string length of the decoded data. If decoding failed, an integer value of 0 is returned.

HaltScanner1

purpose	Stop the scanner port from operating.
syntax	void HaltScanner1 (void);
example call	HaltScanner1( );
description	Use *HaltScanner1* function to stop scanner port from operating. To restart a halted scanner port, the initialization function, *InitScanner1*, must be called. It is recommended that the scanner port should be stopped if the barcode decoding is not required for a long period of time.
return	none

InitScanner1

purpose	Initialize respective scanner port.
syntax	void InitScanner1(void);
example call	InitScanner1( ); while (1) { if (Decode( )) break; }
description	Use *InitScanner1* function to initialize scanner port. The scanner port won't work unless it is initialized.
return	none

2.3 Keyboard Wedge Interface

The portables that equipped with keyboard-wedge interface are able to send data to the host through the wedge interface by using ***SendData*** function. The *SendData* function is governed by a 3-byte unsigned character string -the ***WedgeSetting***, which is a system-defined global character array. User must fill it with appropriate values before calling the *SendData* function.

2.3.1 Definition of the *WedgeSetting* array

Subscript	Bit	Description
0	7 - 0	KBD / Terminal Type
1	7	1 : enable capital lock auto-detection 0 : disable capital lock auto-detection
1	6	1 : capital lock on 0 : capital lock off
1	5	1 : ignore alphabets case 0 : alphabets are case sensitive
1	4 - 3	00 : normal 10 : digits are at lower position 11 : digits are at upper position
1	2-1	00 : normal 10 : capital lock keyboard 11 : shift lock keyboard
1	0	1 : use numeric key pad to transmit digits 0 : use alpha-numeric key to transmit digits
2	7 - 0	inter-character delay

2.3.2 KBD / Terminal Type

The following list shows the possible values of **WedgeSetting[0]**.

Setting Value	Terminal Type	Setting Value	Terminal Type
0	Null (Data not Transmitted)	21	PS55 002-81, 003-81
1	PCAT (USA)	22	PS55 002-2, 003-2
2	PCAT (French)	23	PS55 002-82, 003-82
3	PCAT (German)	24	PS55 002-3, 003-3
4	PCAT (Italian)	25	PS55 002-8A, 003-8A
5	PCAT (Swedish)	26	IBM 3477 TYPE 4
6	PCAT (Norwegian)	27	PS2-30
7	PCAT (UK)	28	Memorex Telex 122 Keys
8	PCAT (Belgium)	29	PCXT
9	PCAT (Spanish)	30	IBM 5550
10	PCAT (Portuguese)	31	NEC 5200
11	PS55 A01-1	32	NEC 9800
12	PS55 A01-2	33	DEC VT220,320,420
13	PS55 A01-3	34	Macintosh (ADB)
14	PS55 001-1	35	Hitachi Elles
15	PS55 001-81	36	Wyse Enhance KBD (US)
16	PS55 001-2	37	NEC Astra
17	PS55 001-82	38	Unisys TO-300
18	PS55 001-3	39	Televideo 965
19	PS55 001-8A	40	ADDS 1010
20	PS55 002-1, 003-1		

2.3.3 Capital Lock Status Setting

To send alphabets with correct case (upper or lower case), the *SendData* routine must know the capital lock status of keyboard when transmitting data. Incorrect capital lock setting will result in different letter case (‘A’ becomes ‘a’, and ‘a’ becomes ‘A’).

2.3.4 Capital Lock Auto-Detection

When the keyboard type selected is either PCAT (all available languages), PS2-30, PS55, or Memorex Telex, *SendData* routine can automatically detect the capital lock status of keyboard when transmitting data, if this setting is enabled. If this is the case, the *SendData* routine will ignore the capital lock status setting and perform auto-detection when transmitting data. If the auto-detection setting is disabled, the *SendData* routine will transmit alphabets according to the setting of the capital lock status.

If the keyboard type selected is neither PCAT, PS2-30, PS55, nor Memorex Telex, the *SendData* routine will transmit the alphabets according to setting of the capital lock status even though the auto-detection setting is enabled.

2.3.5 Alphabets Case

The setting of this bit affect the way *SendData* routine transmits alphabets. The *SendData* routine can transmit alphabets according to their original case (case sensitive) or just ignore it. If ignoring case is selected, the *SendData* routine will always transmit alphabets without adding shift key.

2.3.6 Digits Position

This setting can force the *SendData* routine to treat the position of the digit keys on the keyboard differently. If this setting is set to upper, the *SendData* routine will add shift key when transmitting digits. Please configure this setting to **Normal** unless the user is absolutely sure what he is doing. This setting will be effective only when the keyboard type selected is either PCAT (all available language), PS2-30, PS55, or Memorex Telex. Also if the user choose to send digits using numeric keypad, then this setting is meaningless.

2.3.7 Shift / Capital Lock Keyboard

This setting can force the *SendData* routine to treat the keyboard type to be a shift lock keyboard or a capital lock keyboard. Please configure this setting to **Normal** unless the user is absolutely sure what he is doing. This setting will be effective only when the keyboard type selected is either PCAT (all available language), PS2-30, PS55, or Memorex Telex.

2.3.8 Digit Transmission

This setting instructs the *SendData* routine which group of keys are used to transmit digits, whether to use the digit keys on top of the alphabet keys or use the digit keys on the numeric key pad.

2.3.9 Inter-Character Delay

A 0 to 255 ms inter-character delay can be added before transmit each character. This is used to provide some response time for PC to process keyboard input.

2.3.10 Composition of Output String

The keyboard wedge character mapping is shown below. When the *SendData* routine transmits data, each character in the output string is translated by this table.

	00	10	20	30	40	50	60	70	80
0		F2	SP	0	@	P	`	p	⓪
1	INS	F3	!	1	A	Q	a	q	①
2	DLT	F4	"	2	B	R	b	r	②
3	Home	F5	#	3	C	S	c	s	③
4	End	F6	$	4	D	T	d	t	④
5	Up	F7	%	5	E	U	e	u	⑤
6	Down	F8	&	6	F	V	f	v	⑥
7	Left	F9	'	7	G	W	g	w	⑦
8	BS	F10	(	8	H	X	h	x	⑧
9	HT	F11	)	9	I	Y	i	y	⑨
A	LF	F12	*	:	J	Z	j	z	
B	Right	ESC	+	;	K	[	k	{	
C	PgUp	Exec	,	<	L	\	l	|	
D	CR	CR*	-	=	M	]	m	}	
E	PgDn		.	>	N	^	n	~	
F	F1		/	?	O	_	o	Dly	

Dly :
Delay 100 ms

⓪…⑨ :
Digits of Numeric Key Pad

CR* : Enter key on the numeric key pad

The *SendData* routine can not only transmit simple characters as above, but also provide a way to transmit combination key status, or even direct scan code. This is done by inserting some special command code in the output string. A command code is a character whose value is between 0xC0 and 0xFF.

0xC0 : Indicates that the next character is to be treated as scan code. Transmit it as it is, no translation required.

0xC0 | 0x01 : Send next character with Shift key.

0xC0 | 0x02 : Send next character with left Ctrl key.

0xC0 | 0x04 : Send next character with left Alt key.

0xC0 | 0x08 : Send next character with right Ctrl key.

0xC0 | 0x10 : Send next character with right Alt key.

0xC0 | 0x20 : Clear all combination status key after sending the next character.

For example, to send **[A] [Ctrl-Insert] [5] [scan code 0x29] [Tab] [2] [Shift-Ctrl-A] [B] [Alt-1] [Alt-2-Break] [Alt-1] [Alt-3]**, the following characters are fill into the string supplied to the *SendData* routine when calling. Please note that, the scan code 0x29 is actually a space for PCAT, Alt-12 is a form feed character, and Alt-13 is an ENTER. The **break** after Alt-12 is necessary, if omitted the characters will be treated as Alt-1213 instead of Alt-12 and Alt-13.

0x41, 0xC2, 0x01, 0x35, 0xC0, 0x29, 0x09, 0x32, 0xC3, 0x41, 0x42, 0xC4, 0x31 0xE4, 0x32, 0xC4, 0x31, 0xC4, 0x33, 0x00

SendData

purpose	Send a string to keyboard interface.
syntax	void SendData (char* out_str);
example call	SendData (CodeBuf);
description	*SendData* routine transmits a string pointed by *out_str* to the keyboard interface.
return	None.

WedgeReady	
purpose	Check if the keyboard cable is connected or not.
syntax	int WedgeReady (void);
example call	if (WedgeReady( )) SendData (CodeBuf);
description	Before sending data via keyboard interface, it is recommended to check the cable status first, otherwise the transmission may be blocked.
return	1 if the connection is OK and ready for transmission, 0 otherwise.

2.4 Buzzer

This section describes the beeper manipulation routines. The activating of beeper is directed by specifying a **beeper sequence**, which is a series of **beep frequency** / **beep duration** pairs. Once a beeper sequence is specified, the activation of the beeper is automatically handled by the background operating system. There is no need for the application program to wait for the stop of beeper.

Also there are routines for determining whether a beeper sequence is under going, or to terminate a beeper sequence immediately.

2.4.1 Beeper Sequence

A beeper sequence is an integer array that used to instruct how the beeper activates. It is comprised of **beep frequency** / **beep duration** pairs. Each pair represents one beep. A beep with beep duration value of 0 represents end of beeper sequence, the beeper will then terminate activation.

2.4.2 Beep Frequency

A beep frequency is an integer used to specify the frequency (tone) when the beeper activates. The actual frequency that the beeper activates is not the value specified to the beep frequency. It is calculated by the following formula.

Beep Frequency = 76000 / Actual Frequency Desired

For instance, to get a frequency of 4KHz, the value of beep frequency should be 19. If no sound is desired (pause), the beep frequency should be set to 0. A beep with frequency 0 does not terminate the beeper sequence. Suitable frequency for the beeper ranges from 1 to 6 KHz, where peak at 4 KHz.

2.4.3 Beep Duration

Beep duration is an integer used to specify how long the beeper activates with a specified beep frequency. Beep duration is specified in units of 0.01 second. To get a beep of 1 second, the beep duration should be 100. Beep duration with value of 0 will terminate the beeper sequence.

beeper_status

purpose	To see whether a beeper sequence is under going or not.
syntax	int beeper_status (void);
example call	while (beeper_status( )); /* wait till beeper sequence complete */
description	The *beeper_status* function checks if there is a beeper sequence in progress.
return	1 if beeper sequence still in progress, 0 otherwise

off_beeper

purpose	Terminate beeper sequence.
syntax	void off_beeper (void);
example call	off_beeper ( );
description	The *off_beeper* function terminates beeper sequence immediately if there is a beeper sequence in progress.
return	The *off_beeper* function has no return value.

on_beeper

purpose	Assign a beeper sequence to instruct beeper action.
syntax	void on_beeper (int* sequence); int* sequence; /* pointer to integer array where beeper sequence resides */
example call	int two_beeps[]= { 19, 10, 0, 10, 19, 10, 0, 0 }; on_beeper (two_beeps);
description	The *on_beeper* function assigns a beeper sequence to instruct how the beeper activates. If there is a beeper sequence already in progress, the newly assigned beeper sequence will override the old one.
return	The *on_beeper* function has no return value.

2.5 Calendar

This section describes the calendar manipulation routines. The system date and time are kept by the calendar chip, and they can be retrieved from or set to the calendar chip by the **get_time** and **set_time** functions. A backup rechargeable Lithium battery keeps the calendar chip running even when the power is turned off.

Note that the system time variable *sys_msec*, and *sys_sec* is maintained by CPU timers and has nothing to do with this calendar chip. Accuracy of these two time variables depends on the CPU clock and is not suitable for precise time manipulation. Also, they are reset to 0 upon power up.

2.5.1 Leap Year

The calendar chip automatically handles the leap year. The **year** field set to the calendar chip must be in four-digit year.

DayOfWeek

purpose Get the day of the week information.

syntax int DayOfWeek (void);

example call day = DayOfWeek ( );

description The *DayOfWeek* function returns the day of week information based on current date.

return The *DayOfWeek* function returns an integer indicating the day of week information. A value of 1 to 6 represents Monday to Saturday accordingly. And a value of 7 indicates Sunday.

get_time

purpose Get current date and time.

syntax
```
int get_time (char*cur_time);
char* cur_time;          /*pointer of character array where the date
                           and time will be copied to */
```

example call get_time (system_time);

description The *get_time* function reads current date and time from the calendar chip and copies them to a character array specified in the argument *cur_time.* The character array *cur_time* allocated must have a minimum of 15 bytes to accommodate the date, time, and the string terminator. The format of the system date and time is listed below.

"YYYYMMDDhhmmss"

where **YYYY** : year, 4 digits
MM : month, 2 digits
DD : day, 2 digits
hh : hour, 2 digits
mm : minute, 2 digits
ss : second, 2 digits

return Normally the *get_time* function always returns an integer value of 1. If the calendar chip malfunctions, the *get_time* function will then return 0 to indicate error.

set_time

purpose Set new date and time to the calendar chip.

syntax
```
int set_time (char* new_time);
char* new_time;
```

example call
```
set_time ("19980105125800"); /* JAN 5, 1998  12:58:00 */
```

description The *set_time* function set a new system date and time specified in the argument *new_time* to the calendar chip. The character string *new_time* must have the following format,

"YYYYMMDDhhmmss"

where **YYYY** : year, 4 digits
MM : month, 2 digits, 1-12
DD : day, 2 digits, 1-31
hh : hour, 2 digits, 0-23
mm : minute, 2 digits, 0-59
ss : second, 2 digits, 0-59

return Normally the *set_time* function always returns an integer value of 1. If the calendar chip malfunctions, the *set_time* function will then return 0 to indicate error. Also, if the format is illegal (e.g. set hour to 25), the operation is simply denied and the time is not changed.

2.6 File Manipulation

There are many file manipulation routines available for programming the portable terminals. These routines can help to manipulate the transaction data and ease the implementation of data base system.

There are two types of file structures supported. One is sequential structure called **DAT** file that is usually used to store transaction data. The other is index structure that is usually used to store lookup data. Actually there are two types of index file. One is **DBF** file for storing the original data records (data members), and the other is **IDX** for sorting the records according to the associate key. We will talk about these two file structures in more detail later in this section.

Please note that, not all of the routines described in this section apply to both types of files. In the description paragraph of each routine, please check their applicable file type(s).

2.6.1 File System

On each terminal, there is an on-board data memory (SRAM). This is the place where all the system parameters, program variables, program stack, and file system reside.

2.6.2 File Name

A file name is a null terminated character string with at least 1 and up to 8 characters (not including the null character), which is used to identify the file in the system. There is no file extension as in MS-DOS operation system. The file name is case and if a file name specified is longer than 8 characters, it will be truncated to 8 characters. The file name can be changed later by the *rename* function.

2.6.3 File Handle (File Descriptor)

File handle is the identification of a file after the file is opened. Most of the file manipulation functions need file handles instead of file names when calling them. A file handle is a positive integer (excludes 0) returned from the system when a file is created or opened. All subsequent file operations can then use the file handle to identify the file.

2.6.4 Error Code

There is a system variable "**fErrorCode**" for indicating the result of the last file operation. A value other than 0 indicates error. Also, the error code can be accessed by calling the *read_error_code* function.

2.6.5 Directory

The file system is flat, i.e., it does not support tree-like directory structure and no sub-directory can be created. The maximum number of files supported in the system is limited to 32 files (including all DAT files, DBF files, and their associate IDX files). To get the information of the file directory, you can call the *filelist* routine.

2.6.6 DAT Files

DAT files have a sequential file structure. Data at the beginning of a DAT file can be removed by calling the ***delete_top*** or ***delete_topln*** function. The new file top, the file pointer, and the size of the DAT file will be adjusted accordingly after calling either of the functions. The ***append*** and ***appendln*** functions can write data to the EOF (end of file) position, no matter where the file pointer points to. That is, the file pointer position is not changed after calling these functions. Normally this is the scheme for handling the transaction data, that is, reading and removing data from top of the file, and adding new data to the bottom of a file.

2.6.7 DBF Files and IDX Files

The DBF files and the IDX files form the platform of the data base system. A DBF file has a fixed record length structure. This is the file that stores the data records (members). Whereas, the associate IDX files are the files that keep the information of the position of each record stored in the DBF file, but they are re-arranged (sorted) according to some specific key values.

A library would be a good example to illustrate how DBF and IDX file work. When you are trying to find a specific book in a library, you always start from looking into indexes. The book can be found by looking into the index of **book title**, **writer**, **publisher**, **ISBN number**, …etc. All these indexes are sorted in ascending order for easy lookup according to some specific information of books (book title, writer, publisher, ISBN number, …). When the book is found in the index, it will tell you where the book is actually kept.

As you can see, the books kept in the library are analogous to the data records stored in the DBF file, and the various indexes are just its associate IDX files. Some information in the data records (the book title, writer, publisher, and ISBN number) is used to create the IDX files.

Each DBF file can have at most 8 associate IDX files, and each of them is identified by its key (index) number. The key number is assigned by user program when the IDX file is created. The valid key numbers are from 1 to 8.

Data records are not fetched directly from the DBF file but rather through associate IDX files. The value of file pointers of the IDX files (index pointers) does not represent the address of the data records stored in the DBF file. It indicates the sequence number of the specific data record in the IDX file.

access

applicable file DAT DBF

purpose Check for file existence.

syntax
```
int access (char* filename);
char* filename;          /* file name of the file being checked */
```

example call if (access("data1")) puts("data1 exist!\n");

description Check if the file specified by *filename* exists. If *filename* exceeds 8 characters, it will be truncated to 8 characters.

return If the file specified by *filename* exist, *access* returns an integer value of 1, 0 otherwise. In case of error, *access* will return an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
1	*filename* is a NULL string.

add_member

applicable file DBF

purpose Add a data record (member) to a DBF file.

syntax
```
int add_member (int DBF_fd, char* member);
int DBF_fd;              /* file handle of target DBF file */
char* member;            /* pointer to a character array from where
                            the added member is copied */
```

example call add_member(DBF_fd, member);

description The *add_member* function adds a member specified by the argument *member* to a DBF file whose file handle is *DBF_fd* and add index entries to all the IDX file associated to it. If the length of the added member is greater than the length defined for the DBF file (*member_len* in *create_DBF* function), the member will be truncated to that length.

return If *add_member* successfully adds the member, it returns an integer value of 1. In case of error, *add_member* will return an integer value of 0 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by *DBF_fd* does not exist.
4	File specified by *DBF_fd* is not a DBF file.
7	Invalid file handle
8	File not opened
10	No free file space for adding member.

append

applicable file DAT

purpose Write a specified number of bytes to bottom (end-of-file position) of a DAT file.

syntax
```
int append (int fd, char* buffer, int count);
int fd;          /* file handle of the target DAT file */
char* buffer;    /* pointer to array of characters representing data to be
                    written */
```

int count; **/*** number of bytes to be written */

example call append(fd, “1234567890”, 10);

description The *append* function writes the number of bytes specified in the argument *count* from the character array *buffer* to the bottom of a DAT file whose file handle is *fd*. Writing of data starts at the end-of-file position of the file, and the file pointer position is unaffected by the operation. The *append* function will automatically extend the file size of the file to hold the data written.

return The *append* function returns the number of bytes actually written to the file. In case of error, *append* returns an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by *fd* does not exist.
4	File specified by *fd* is not a DAT file.
7	Invalid file handle
8	File not opened
9	The value of *count* is negative.
10	No more free file space for file extension.

comments The maximum number of characters can be written is limited to 32767.

appendln

applicable file DAT

purpose Write a null terminated character string to the bottom (end-of-file position) of a DAT file.

syntax
```
int appendln (int fd, char* buffer);
int fd;          /* file handle of the target DAT file */
char* buffer;    /* pointer to array of characters representing data to be
                    written */
```

example call appendln (fd, data_buffer);

description The *appendln* function writes a null terminated character string from the character array *buffer* to a DAT file whose file handle is *fd*. Characters are written to the file until a null character (\0) is encountered. The null character is also written to the file. Writing of data starts at the end-of-file position. The file pointer position is unaffected by the operation. The *appendln* function will automatically extend the file size of the file to hold the data written.

return The *appendln* function returns the number of bytes actually written to the file (includes the null character). In case of error, *appendln* returns an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by *fd* does not exist.
4	File specified by *fd* is not a DAT file.
7	Invalid file handle
8	File not opened
10	No more free file space for file extension.
11	Can not find string treminator in *buf*.

comments The maximum number of characters can be written is limited to 32767.

chsize

applicable file DAT

purpose Extends or truncates a DAT file.

syntax
```
int chsize (int fd, long new_size);
int fd;                 /* file handle of the target DAT file */
long new_size;          /* new length of file in bytes */
```

example call
```
if (chsize(fd,0L)) puts("file truncated!\n");
```

description The *chsize* function truncates or extends the file specified by the argument *fd* to match the new file length in bytes given in the argument *new_size.* If the file is truncated, all data beyond the new file size will be lost. If the file is extended, no initial value is filled to the newly extended area.

return If *chsize* successfully changes the file size of the specified DAT file, it returns an integer value of 1. In case of error, *chsize* will return an integer value of 0 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by *fd* does not exist.
4	File specified by *fd* is not a DAT file.
7	Invalid file handle
8	File not opened
10	No more free file space for file extension.

close

applicable file DAT

purpose Close a DAT file.

syntax
```
int close(int fd);
int fd;                 /* file handle of the target DAT file */
```

example call
```
if (close(fd)) puts("file closed!\n");
```

description Close a previously opened or created DAT file whose file handle is *fd*.

return *close* returns an integer value of 1 to indicate success. In case of error, *close* returns an integer value of 0 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by *fd* does not exist.
4	File specified by *fd* is not a DAT file.
7	Invalid file handle
8	File not opened

close_DBF

applicable file DBF

purpose Close DBF and its associated IDX file.

syntax
```
int close_DBF (int DBF_fd);
int DBF_fd;             /* file handle of the target DBF file */
```

example call
```
if (close_DBF(DBF_fd)) send_lcds("DBF file closed!\n");
```

description Close a previously opened or created DBF file whose file handle is *DBF_fd*. The *close_DBF* function not only closes the specified DBF file but also closes all the IDX files associated to it.

return The *close_DBF* function returns an integer value of 1 to indicate success. In case of error, *close_DBF* returns an integer value of 0 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by *DBF_fd* does not exist.
4	File specified by *DBF_fd* is not a DBF file.
7	Invalid file handle
8	File not opened

create_DBF

applicable file DBF

purpose Create a DBF file and get the file handle of the file for further processing.

syntax
```
int create_DBF (char* filename, unsigned member_len);
char* filename;          /* file name of the DBF file being created */
unsigned member_len;     /* member (record) length of the DBF file */
```

example call
```
if (fd = create_DBF("data1",64) > 0) puts("data1 created!\n");
```

description The *create_DBF* function creates a DBF file specified by *filename* and gets the file handle of the file. A file handle is a positive integer (excludes 0) used to identify the file for subsequent file manipulations on the file. The argument *member_len* supplied in the function call specifies the maximum member length for the DBF file. Any members subsequently added to this DBF file with length greater than *member_len* will be truncated to this length. If *filename* exceeds 8 characters, it will be truncated to 8 characters.

return If *create_DBF* successfully creates the DBF file, it returns the file handle of the file being created. In case of error, *create_DBF* will return an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
1	*filename* is a NULL string.
6	Can't create file. Because the maximum number of files allowed in the system is exceeded.
9	Illegal argument : *member_len*
12	File specified by *filename* already exists.

create_index

applicable file DBF

purpose Create an IDX file of a DBF file.

Syntax
```
int create_index (int DBF_fd, int key_number, int key_offset, int key_len);
int DBF_fd;          /* file handle of a DBF file which the target index
                        file associated to  */
int key_number;  /*key number of the index file to be created */
int key_offset;    /* the byte offset address in member where the key
```

value begins */
int key_len; /* the length (size of) of key value for the index */

example call create_index (DBF_fd,1,0,10);

description The *create_index* function creates an IDX file specified by the argument *key_number* which is associated to a DBF file whose file handle is *DBF_fd*. The key value field for the index is specified by the argument *key_offset* and *key_len*. The argument *key_offset* specifies the byte offset address where the key value in a member begins. And *key_len* specifies the length of the key value. The key field defined by *key_offset* and *key_len* should be within the member as defined by *member_len* in *create_DBF* function. That is, *key_offset* plus *key_len* should not greater than *member_len*. The *create_index* function can only be called before any members are added to the DBF file. That is, when the DBF file is empty (no members exist). If any member should exist in the DBF file, *rebuild_index* should be used instead.

return If *create_index* successfully creates an IDX file, it returns an integer value of 1. In case of error, *create_index* will return an integer value of 0 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by *DBF_fd* does not exist.
4	File specified by *DBF_fd* is not a DBF file.
6	Can't create file. Because the maximum number of files allowed in the system is exceeded.
7	Invalid file handle
8	File not opened
13	Illegal value in argument *key_number.*
17	Illegal value in argument *key_offset*, and/or *key_len*.
18	DBF file specified by *DBF_fd* is not empty.
19	IDX file specified by *key_number* already exists.

delete_member

applicable file DBF

purpose Delete a member of a DBF file.

syntax
int delete_member (int DBF_fd, int key_number);
int DBF_fd; /* file handle of target DBF file ***/***
int key_number; /* key number of the index file whose index pointer points to the target member */

example call delete_member (DBF_fd, 1);

description The *delete_member* function deletes the member pointed by the index pointer of an IDX file whose key number is specified in the argument *key_number*. The DBF file which the IDX file associates to is specified in the argument *DBF_fd*.

return If *delete_member* successfully deletes the member, it returns an integer value of 1. In case of error, *delete_member* will return an integer value of 0 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by *DBF_fd* does not exist.
4	File specified by *DBF_fd* is not a DBF file.
7	Invalid file handle
8	File not opened
13	Illegal value in argument *key_number.*
14	The IDX file specified by *key_number* does not exist.
16	There are no members in the DBF file.

delete_top

applicable file DAT

purpose Remove a specified number of bytes from top (beginning-of-file position) of a DAT file.

syntax
```
int delete_top (int fd, int count);
int fd;          /* file handle of the target DAT file */
int count;       /* number of bytes to be removed */
```

example call
```
delete_top (fd, 80);
```

description The *delete_top* function removes the number of bytes specified in the argument *count* from a DAT file whose file handle is *fd*. Removing of data starts at the beginning-of-file position of the file. The file pointer position is adjusted accordingly by the operation. For instance, if initially the file pointer points to the tenth character, after removing 8 character from the file, the new file pointer will points to the second character of the file. The *delete_top* function will resize the file size automatically.

return The *delete_top* function returns the number of bytes actually removed from the file. In case of error, *delete_top* returns an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by *fd* does not exist.
4	File specified by *fd* is not a DAT file.
7	Invalid file handle
8	File not opened
9	The value of *count* is negative.

delete_topln

applicable file DAT

purpose Remove a null terminated character string from the top (beginning-of-file position) of a DAT file.

syntax
```
int delete_topln (int fd);
int fd;          /* file handle of the target DAT file */
```

example call
```
delete_topln (fd);
```

description The *delete_topln* function removes a line terminated by a null character from a DAT file whose file handle is *fd*. Characters are removed from the file until a null character (\0) or end-of-file is encountered. The null character is also removed from the file. Removing of data starts at the top (beginning-of-file position) of the file, and the file pointer position is adjusted accordingly. The *delete_topln* function will resize the file size automatically.

return The *delete_topln* function returns the number of bytes actually removed from the file (includes the null character). In case of error, *delete_topln* returns an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by *fd* does not exist.
4	File specified by *fd* is not a DAT file.
7	Invalid file handle
8	File not opened

eof

applicable file DAT

purpose Check if file pointer of a DAT file reaches end of file.

syntax
```
int eof (int fd);
int fd;          /* file handle of the target DAT file */
```

example call
```
if (eof(fd)) puts("end of file reached!\n");
```

description The *eof* function checks if the file pointer of the DAT file whose file handle is specified in the argument *fd*, points to end-of-file.

return The *eof* function returns an integer value of 1 to indicate an end-of-file and a 0 when not. In case of error, *eof* returns an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by *fd* does not exist.
4	File specified by *fd* is not a DAT file.
7	Invalid file handle
8	File not opened

filelength

applicable file DAT

purpose Get file length information of a DAT file.

syntax
```
long filelength (int fd);
int fd;          /* file handle of the target DAT file */
```

example call
```
data_size = filelength (fd);
```

description The *filelength* function returns the size in number of bytes of the DAT file whose file handle is specified in the argument *fd*.

return The long integer value returned by *filelength* is the size of the DAT file in number of bytes. In case of error, *filelength* returns a long value of -1L and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by *fd* does not exist.
4	File specified by *fd* is not a DAT file.
7	Invalid file handle
8	File not opened

filelist

purpose Get file directory information.

syntax
```
int filelist (char* dir);
char* dir;        /* pointer to a character array where the file directory
                     information is copied to */
```

example call
```
total_file = filelist (dir);
```

description The *filelist* function copies the file name, file type, and file size information (separated by a blank character) of all files in existence into a character array specified in the argument *dir*.

return The *filelist* function returns the number of files currently exist in the system.

get_member

applicable file DBF

purpose Read the member pointed by the index pointer.

syntax
```
int get_member (int DBF_fd, int key_number, char* buffer);
int DBF_fd;       /* file handle of a DBF file which the target index
                     file associated to */
int key_number; /* key number of the target index file
char* buffer;     /* pointer to a character array where the member is
                     copied to */
```

example call
```
if (get_member(DBF_fd,1,buffer) == 0) puts(buffer);
```

description The *get_member* function copies the member pointed to by a index pointer to a character array specified in the argument *buffer*. The IDX file concerned is specified in the argument *key_number* which is associated to a DBF file whose file handle is *DBF_fd*.

Return The *get_member* function returns an integer value of 1 to indicate success. In case of error, *get_member* returns an integer value of 0 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by *DBF_fd* does not exist.
4	File specified by *DBF_fd* is not a DBF file.
7	Invalid file handle
8	File not opened
13	Illegal value in argument *key_number.*
14	The IDX file specified by *key_number* does not exist.
16	There are no members in the DBF file.

has_member

applicable file DBF

purpose Check if a specific member exist in an IDX file.

syntax
```
int has_member (int DBF_fd, int key_number, char* key_value);
int DBF_fd;       /* file handle of a DBF file which the target index
                     file associated to */
int key_number;   /* key number of the target index file */
```

```
char* key_value; /* pointer of a character array which is used to
                    identify a specific member */
```

example call

```
if (has_member(DBF_fd,1,"WANG"))
        puts("WANG is on the name list!\n");
```

description The *has_member* function tries to locate a member which matches the key value specified in the argument *key_value* in an IDX file *key_number*. The IDX file is associated to a DBF file whose file handle is specified in the argument *DBF_fd*. If there is a complete match to the *key_value*, the index pointer will point to the first of all matches. In case there are several members with the same key value, the user can then check each member sequentially from the member pointed by the index pointer to find the desired member. If *has_member* does not find a complete match in the index, the index pointer will still point to the first member with key value greater than *key_value* specified.

return The *has_member* function returns an integer value of 1 to indicate a complete match in key value has been found, 0 if not. In case of error, *has_member* returns an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by *DBF_fd* does not exist.
4	File specified by *DBF_fd* is not a DBF file.
7	Invalid file handle
8	File not opened
13	Illegal value in argument *key_number.*
14	The IDX file specified by *key_number* does not exist.

lseek

applicable file DAT

purpose Move file pointer of a DAT file to a new position.

syntax

```
long lseek (int fd, long offset, int origin);
int fd;         /* file handle of the target DAT file */
long offset;    /* offset of new position (in bytes) from origin */
int origin;     /* constant indicating the position from where to offset */
```

example call

```
lseek(fd, 512L, 0);             /* skip 512 bytes */
```

description The *lseek* function moves the file pointer of a DAT file whose file handle is specified in the argument *fd* to a new position within the file. The new position is specified with an offset byte address to a specific origin. The offset byte address is specified in the argument *offset* which is a long integer. There are 3 possible values for the argument *origin*. The values and their interpretations are listed below.

Value of origin	Interpretation
1	beginning of file
0	current file pointer position
-1	end of file

return When successful, *lseek* returns the new byte offset address of the file pointer from the beginning of file. In case of error, *lseek* returns a long value of -1L and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by *fd* does not exist.
4	File specified by *fd* is not a DAT file.
7	Invalid file handle
8	File not opened
9	Illegal *origin* value.
15	New position is beyond end-of-file.

lseek_DBF

applicable file DBF

purpose Move index pointer of an IDX file to a new position.

syntax
```
long lseek_DBF (int DBF_fd, int key_number, long offset, int origin);
int DBF_fd; /* file handle of a DBF file which the target index file
               associated to */
int key_number; /* key number of the target index file */
long offset;    /* offset of new position, sequence number from origin */
int origin;     /* constant indicating the position from where to offset */
```

example call
```
lseek_DBF(DBF_fd, 1, 1L, 0);  /* move to next member */
```

description The *lseek_DBF* function moves the index pointer of a INDEX file which is specified in the argument *key_number* to a new position. The index file is associated to a DBF file whose file handle is in the argument *DBF_fd*. The new position is specified with an offset sequence address to a specific origin. The offset rank address is specified in the argument *offset* which is a long integer. There are 3 possible values for the argument *origin*. The values and their interpretations are listed below.

Value of origin	Interpretation
1	first index of index file
0	current index pointer position
-1	last index of index file

return When successful, *lseek_DBF* returns the new sequence position that the index pointer points to. In case of error, *lseek_DBF* returns a long value of -1L and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by *DBF_fd* does not exist.
4	File specified by *DBF_fd* is not a DBF file.
7	Invalid file handle
8	File not opened
9	Illegal *origin* value.
13	Illegal value in argument *key_number.*
14	The IDX file specified by *key_number* does not exist.
15	New position is beyond end-of-file.

member_in_DBF

applicable file DBF

purpose Determine how many members exist in a DBF file.

syntax
```
long member_in_DBF (int DBF_fd);
int DBF_fd;     /* file handle of the target DBF file */
```

example call
```
total_member = member_in_DBF(DBF_fd);
```

description The *member_in_DBF* function returns the number of member in a DBF file whose file handle is specified in the argument *DBF_fd*.

return The long integer value returned by *member_in_DBF* is the number of members exist in the DBF file. In case of error, *member_in_DBF* returns a long value of -1L and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by *DBF_fd* does not exist.
4	File specified by *DBF_fd* is not a DBF file.
7	Invalid file handle
8	File not opened

open

applicable file DAT

purpose Open a DAT file and get the file handle of the file for further processing.

Syntax int open (char* filename);
char* filename; /* file name of file to be opened */

example call if (fd = open("data1") > 0) puts("data1 opened!\n");

description The *open* function opens a DAT file specified by *filename* and gets the file handle of the file. A file handle is a positive integer (excludes 0) used to identify the file for subsequent file manipulations on the file. If the file specified by *filename* does not exist, it will be created first. If *filename* exceeds 8 characters, it will be truncated to 8 characters long. After the file is opened, the file pointer points to the beginning of file.

return If *open* successfully opens the file, it returns the file handle of the file being opened. In case of error, *open* will return an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
1	*filename* is a NULL string.
4	File specified by *filename* is not a DAT file.
5	File specified by *filename* is already opened.
6	Can't create file. Because the maximum number of files allowed in the system is exceeded.

open_DBF

applicable file DBF

purpose Open a DBF file and get the file handle of the file for further processing.

syntax int open_DBF (char* filename);
char* filename; /* file name of file to be opened */

example call if (fd = open_DBF("data1") > 0) puts("data1 opened!\n");

description The *open_DBF* function opens a DBF file specified by *filename* and gets the file handle of the file. A file handle is a positive integer (excludes 0) used to identify the file for subsequent file manipulations on the file. The *open_DBF* function will also open all the index (key) files associated to the DBF file being opened simultaneously. If *filename* exceeds 8 characters, it will be truncated to 8 characters long. After the DBF file is

opened, the index pointers of all the associated index (key) files point to the beginning of the respective index.

return If *open_DBF* successfully opens the DBF file, it returns the file handle of the file being opened. In case of error, *open_DBF* will return an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
1	*filename* is a NULL string.
2	File specified by *filename* does not exist.
4	File specified by *filename* is not a DBF file.
5	File specified by *filename* is already opened.

read

applicable file DAT

purpose Read a specified number of bytes from a DAT file.

syntax
```
int read (int fd, char* buffer, unsigned count);
int fd;              /* file handle of the target DAT file */
char* buffer;        /* pointer to array of characters where the read data
                        will be placed */
unsigned count;   /* number of bytes to be read */
```

example call
```
if ((bytes_read = read(fd, buffer,80)) == -1)
        puts("read error!\n");
```

description The *read* function copies the number of bytes specified in the argument *count* from the DAT file whose file handle is *fd* to the array of characters *buffer*. Reading starts at the current position of the file pointer, which is incremented accordingly when the operation is completed.

return The *read* function returns the number of bytes actually read from the file. In case of error, *read* returns an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
4	File specified by *fd* is not a DAT file.
7	*fd* is not a file handle of a previously opened file.

comments Since *read* returns an signed integer, the return value should be converted to *unsigned int* when reading more than 32,767 bytes of data from a file or the return value will be negative. Because the number of bytes to be read is specified in an unsigned integer argument, you could theoretically read 65,535 bytes at a time. But 65,535 (or FFFFh) also means -1 in signed representation, so when reading 65,535 bytes the return value indicates an error. The practical maximum then is 65,534.

read_error_code

purpose Get the value of the global variable *fErrorCode*.

syntax
```
int read_error_code( );
```

example call
```
if (read_error_code() = = 2) puts("File not exist!\n");
```

description The *read_error_code* function gets the value of the global variable *fErrorCode* and returns the value to the calling program. The programmer can use this function to get the error code of the file manipulation routine previously called. However, the global variable *fErrorCode* can be directly accessed without making a call to this function.

return The *read_error_code* function returns the value of the global variable *fErrorCode*.

readln

applicable file DAT

purpose Read a line terminated by a null character from a DAT file.

syntax
```
int readln(int fd, char* buffer, unsigned max_count);
int fd;              /* file handle of the target DAT file */
char* buffer;    /* pointer to array of characters where the read line will
                       will be placed */
unsigned max_count; /* maximum number of bytes to be read before
                              null character encountered */
```

example call readln(fd, buffer,80);

description The *readln* function reads a line from the DAT file whose file handle is *fd* and stores the characters in the character array *buffer*. Characters are read until end-of-file encountered, a null character (\0) encountered, or the total number of characters read equals the number specified in *max_count*. The *readln* function then returns the number of bytes actually read from the file. The null character (\0) is also counted if read. If the *readln* function completes its operation not because a null character is read, there will be no null character stored in *buffer*. Reading starts at the current position of the file pointer, which is incremented accordingly when the operation is completed.

return The *readln* function returns the number of bytes actually read from the file (includes the null character if read). In case of error, *readln* returns an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
4	File specified by *fd* is not a DAT file.
7	*fd* is not a file handle of a previously opened file.

comments Since *readln* returns an signed integer, the return value should be converted to *unsigned int* when reading more than 32,767 bytes of data from a file or the return value will be negative. Because the number of bytes to be read is specified in an unsigned integer argument, you could theoretically read 65,535 bytes at a time. But 65,535 (or FFFFh) also means -1 in signed representation, so when reading 65,535 bytes the return value indicates an error. The practical maximum then is 65,534. The argument *max_count* is usually set to a value which equals the size of the character array *buffer* to avoid string overflow.

cautions Under some situations (end-of-file encountered or *max_count* reached), there might not be a null character exist in *buffer*.

rebuild_index

applicable file DBF

purpose Rebuild an IDX file of a DBF file.

syntax

```
int rebuild_index (int DBF_fd, int key_number, int preference_index,
                   int key_offset, int key_len);
int DBF_fd;          /* file handle of a DBF file which the target
                        index file associated to */
int key_number;      /* key number of the index file to be created */
int preference_index; /* key number of the preference index file, see
                        description below */
int key_offset;      /* the byte offset address in member where the
                        key value begins */
int key_len;         /* the length (size of) of key value for the index */
```

example call rebuild_index(DBF_fd,1,0,10);

description The *rebuild_index* function rebuilds or creates an index file specified by the argument *key_number* which is associated to a DBF file whose file handle is *DBF_fd*. If the index file specified by *key_number* exists, it will be overwritten, otherwise, the *rebuild_index* function will create and build a new IDX file. The key-value field of the index is specified by the *key_offset* and *key_len* arguments. The argument *key_offset* specifies the byte offset where the key value in a member begins. And *key_len* specifies the length of the key value. The key field defined by *key_offset* and *key_len* should be within the member as defined by *member_len* in *create_DBF* function. That is, *key_offset* plus *key_len* should not greater than *member_len*.

The *rebuild_index* function can be used whenever an index file has same key values in a key field. The argument *preference_index* specifies the index file from which the *rebuild_index* function takes as the input sequence for building the index file. For instance, if a report is to be generated by the sequence of date, department, and ID number. And the date data and department data may be duplicated. Then this can be done by rebuilds the ID number index first and then rebuilds the department index with ID number as the preference index, and finally rebuilds the date index with department index as the preference index. The resulting member sequence in the date index will be in date, department, and ID number. If there is no preferred index desired, the *preference_index* should be 0. The preferred sequence will be the original member sequence in the DBF file.

return If *rebuild_index* successfully creates / rebuilds an IDX file, it returns an integer value of 0. In case of error, *rebuild_index* will return an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
4	File specified by *DBF_fd* is not a DBF file.
6	Can't create file. Because the maximum number of files allowed in the system is exceeded.
8	*DBF_fd* is not a file handle of a previously opened file.
9	Illegal value in argument *key_offset*,and/or *key_len.*
10	No more free file space for rebuilding index.
11	Illegal value in argument *key_number.*
18	Illegal value in argument *preference_index*.

receive_file **720**

applicable file DAT DBF

purpose To receive files from host PC and then store the received files on Smart-Media card.

syntax
```
int receive_file (int com_port, const char *file_name);
int com_port;       /* the COM port from which files will be received */
char *file_name;    /* file name to be used when saving the file */
```

example call
```
open_com (1, 0x08);
rtn_val = receive_file (1, "");
```

description This routine can be used to receive files from host PC or devices that support Z-modem transmission protocol, and the received files will be saved on Smart-Media card. If the file name is not given, the original file name will be used; otherwise, the given file name will replace the original file name.

return If no error found, it returns 0; otherwise, it returns -1.

remove

applicable file DAT DBF

purpose Delete file.

syntax
```
int remove(char* filename);
char* filename; /* file name of file to be deleted */
```

example call
```
if (remove("data1")) puts("data1 deleted!\n");
```

description Delete the file specified by *filename*. If *filename* exceeds 8 characters, it will be truncated to 8 characters long. If the file to be deleted is a DBF file, the DBF file and all the index (key) files associated to it will be deleted altogether.

return If *remove* deletes the file successfully, it returns an integer value of 1. In case of error, *remove* will return an integer value of 0 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretations are listed below.

Error Code	Interpretation
1	*filename* is a NULL string.
2	File specified by *filename* does not exist.

remove_DBF **720**

applicable file DBF on SMC

purpose Delete a DBF file.

syntax
```
remove_DBF(const char* file_name);
const char *filename;          /* name of DBF file to be deleted */
```

example call
```
if (remove_DBF("dbf1")) puts ("dbf1 deleted!\n");
```

description Delete the DBF file specified by *filename,* the DBF file and all the index (key) files associated to it will be deleted altogether. If *filename* exceeds 8 characters, it will be truncated to 8 characters long.

return If *remove_DBF* deletes the file successfully, it returns an integer value of 1. In case of error, *remove_DBF* will return an integer value of 0 and an error code is set to the global variable *fErrorCode* to indicate the error

condition encountered. Possible error codes and their interpretations are listed below.

Error Code	Interpretation
1	*filename* is a NULL string.
2	File specified by *filename* does not exist.
4	The *filename* is not a DBF file.

remove_index

applicable file DBF

purpose Delete an index file.

syntax
```
int remove_index(int DBF_fd, int key_number);
int DBF_fd;       /* file handle of a DBF file which the target index
                     file associated to */
int key_number;/* key number of the target index file */
```

example call
```
if (remove_index(DBF_fd, 1)) puts ("index removed!\n");
```

description The *remove_index* function deletes the index file specified in the argument *key_number* which is associated to a DBF file whose file handle is *DBF_fd.*

return The *remove_index* function returns an integer value of 1 if it successfully deletes the index file. In case of error, *remove_index* returns an integer value of 0 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
4	File specified by *fd* is not a DBF file.
8	*fd* is not a file handle of a previously opened file.
11	Index file specified by *key_number* does not exist.

rename

applicable file DAT DBF

purpose Change file name of an existing file.

syntax
```
int rename(char* old_filename, char* new_filename);
char* old_filename;    /* file name of file to be renamed */
char* new_filename;  /* new file name desired */
```

example call
```
if (rename("data1", "text1")) puts("data1 renamed!\n");
```

description Change the file name of the file specified by *old_filename* to *new_filename*. If either *old_filename* or *new_filename* exceeds 8 characters, it will be truncated to 8 characters long. If the file specified by *old_filename* is a DBF file, the file name of the DBF file and all the index (key) files associated to it will be changed to *new_filename* altogether.

return If *rename* successfully changes the file name, it returns an integer value of 1. In case of error, *rename* will return an integer value of 0, and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
1	Either *old_filename* or *new_filename* is a NULL string.
2	File specified by *old_filename* does not exist.
3	A file with file name *new_filename* already exists.

send_file **720**

applicable file DAT DBF

purpose Send data files to host PC or devices that support Z-modem transmission protocol.

syntax
```
int send_file (int com_port, const char *file_name);
int com_port;       /* the COM port to which files will be sent through*/
char *file_name;    /* name of the to-be-sent file */
```

example call
```
open_com (1, 0x08);
rtn_val = send_file (1, "A:MyData.dat");
```

description This routine can be used to send files to host PC or devices that support Z-modem transmission protocol. Note that the files to be sent must exist on Smart-Media card.

return If no error found, it returns 0; otherwise, it returns -1.

tell

applicable file DAT

purpose Get file pointer position of a DAT file.

syntax
```
long tell (int fd);
int fd;      /* file handle of the target DAT file */
```

example call
```
current_position = tell (fd);
```

description The *tell* function returns the current file pointer position of the DAT file whose file handle is specified in the argument *fd*. The file pointer position is expressed in number of bytes from the beginning of file. For instance, if the file pointer points to the beginning of file, the file pointer position will be 0L.

return The long integer value returned by *tell* is the current file pointer position in file. In case of error, *tell* returns a long value of -1L and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
4	File specified by *fd* is not a DAT file.
7	*fd* is not a file handle of a previously opened file.

tell_DBF

applicable file IDX

purpose Get index pointer position of an IDX file.

syntax long tell_DBF (int DBF_fd, int key_number);
int DBF_fd; /* file handle of the target DAT file */
int key_number;/* key number of the target index file */

example call rank_number = tell_DBF(DBF_fd, 1);

description The *tell_DBF* function returns the current index pointer position of the IDX file which is specified in the argument *key_number*. The IDX file is associated to a DBF file whose file handle is specified in the argument *DBF_fd*. The index pointer position is expressed in rank number in the IDX file. For instance, if the index pointer points to the first index, the index pointer position will be 1L.

return The long integer value returned by *tell_DBF* is the current index pointer position in ranks in file. In case of error, *tell_DBF* returns a long value of -1L and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
4	File specified by *DBF_fd* is not a DAT file.
8	*DBF_fd* is not a file handle of a previously opened file.
11	Index file specified by *key_number* does not exist.

update_member

applicable file DBF

purpose Update a member of a DBF file.

syntax int update_member (int fd, int key_number, char* buffer);
int fd; /* file handle of target DBF file */
int key_number; /* key number of the index file whose index pointer points to the target member */
char* buffer; /* pointer to array of characters representing data to be written */

example call update_member (DBF_fd, 1,1);

description The *update_member* function updates the member pointed by the index pointer of an IDX file whose key number is specified in the argument *key_number*. The DBF file which the IDX file associates to is specified in the argument *DBF_fd*.

return If *update_member* successfully updates the member, it returns an integer value of 1. In case of error, *update_member* will return an integer value of 0. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by *DBF_fd* does not exist.
4	File specified by *DBF_fd* is not a DBF file.
7	Invalid file handle
8	File not opened
13	Illegal value in argument *key_number.*
14	The IDX file specified by *key_number* does not exist.
16	There are no members in the DBF file.

write

applicable file DAT

purpose Write a specified number of bytes to a DAT file.

syntax
```
int write (int fd, char* buffer, unsigned count);
int fd;              /* file handle of the target DAT file */
char* buffer;        /* pointer to array of characters representing data to be
                        written */
unsigned count;      number of bytes to be written
```

example call
```
write (fd, data_buffer, 1024);
```

description The *write* function writes the number of bytes specified in the argument *count* from the character array *buffer* to a DAT file whose file handle is *fd*. Writing of data starts at the current position of the file pointer, which is incremented accordingly when the operation is completed. If the end-of-file condition is encountered during the operation, the file will be extended automatically to complete the operation.

return The *write* function returns the number of bytes actually written to the file. In case of error, *write* returns an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
4	File specified by *fd* is not a DAT file.
7	*fd* is not a file handle of a previously opened file.
10	No more free file space for file extension.

writeln

applicable file DAT

purpose Write a line terminated by a null character (\0) to a DAT file.

syntax
```
int writeln (int fd, char* buffer);
int fd;              /* file handle of the target DAT file */
char* buffer;        /* pointer to array of characters representing data to be
                        written */
```

example call
```
writeln (fd, data_buffer);
```

description The *writeln* function writes a line terminated by a null character from the character array *buffer* to a DAT file whose file handle is *fd*. Characters are written to the file until a null character (\0) is encountered. The null character is also written to the file. Writing of data starts at the current position of the file pointer, which is incremented accordingly when the operation is completed. If the end-of-file condition is encountered during the operation, the file will be extended automatically to complete the operation.

return The *writeln* function returns the number of bytes actually written to the file (includes the null character). In case of error, *writeln* returns an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
4	File specified by *fd* is not a DAT file.
7	*fd* is not a file handle of a previously opened file.
9	no null character found in *buffer*
10	No more free file space for file extension.

2.7 LED

The LED indicators on the portable terminals are usually used to indicate the system status, like good read or bad read, error operations, etc. There are two LEDs on each terminal, namely the red one and the green one. The IDs for each LED are listed below,

Name	Number
LED_RED	0
LED_GREEN	1

set_led

purpose To set the LED indicators

syntax
```
int set_led (int led, int mode, int duration);
int led;        /* number of LED to be accessed */
int mode;       /* activation mode */
int duration;   /* duration in unit of 10 mini-seconds */
```

example call
```
set_led (LED_RED, LED_FLASH, 50);  /* set Red LED to flash for each
                                      1 second cycle*/
```

description 3 modes are supported,

- LED_OFF : off for (duration X 0.01) seconds then on
- LED_ON : on for (duration X 0.01) seconds then off
- LED_FLASH : flash, turn on then turn off the LED for (duration X 0.01) seconds and then repeat

return none

2.8 Keypad

A scanning circuitry of 4 by 8 matrix is utilized on the keyboard of the portable terminals. The background routine constantly scans the keyboard to check if any key is being pressed. There is a keyboard buffer of size 32 bytes. However, if the buffer is full, the keystrokes followed will be ignored. Normally a C program needs constantly to check if any keystroke is available in the buffer.

CheckKey **8000 8300**

purpose To detect if the specified keys have been pressed simultaneously or not.

syntax int CheckKey (const int scan_code,…);

Specify the scan codes of the keys as many as you like, but be sure to specify the type as the last parameter. There are to types:

CHK_EXC /* exclusive checking, only the keys being pressed match the keys specified will the function return 1 */

CHK_INC /* inclusive checking, as long as the keys being pressed Include the keys specified, this function will return 1 */

example call

```
while (1)
    {
    if ( CheckKey (SC_1, SC_2, SC_3, CHK_EXC))
        printf ("The user presses 1, 2, 3 simultaneously");
    OSTimeDly (8);              // delay 8x5 = 40ms
    }
```

description The *CheckKey* function scans the keyboard to check if the specified keys are being pressed or not. Usually this is used to detect special key combinations for a special purpose. Note it may need up to 40ms for the system to scan the whole keyboard, therefore two consecutive calls should not be made within 40ms. If you are not sure how long it may take to run your code in-between two calls, you may call the OSTimeDly function to ensure that the delay is enough.

return 1 if successful, 0 otherwise.

clr_kb

purpose To clear the keyboard buffer.

syntax void clr_kb (void);

example call clr_kb ( );

description The *clr_kb* function clears the keyboard buffer. This function is automatically called by the system upon powering up the terminal.

return none

dis_alpha

purpose Disable the processing of alphabet keystroke.

syntax void dis_alpha (void);

example call dis_alpha ( );

description The *dis_alpha* function disables the alphabet key stroke processing. If the alpha lock status is on prior to calling this function, it will become off after calling this function.

return none

dis_shift	**720**

purpose Disable the processing of shift keystroke.

syntax void dis_shift (void);

example call dis_shift ( );

description The *dis_shift* function disables the shift key stroke processing. If the shift lock status is on prior to calling this function, it will become off after calling this function.

return none

en_alpha

purpose Enable the processing of alphabet keystroke.

syntax void en_alpha (void); // 711, 720, 8100
void en_alpha (int type); // 8000, 8300
int type; // key rolling type

example call en_alpha ( );

description The alphabet keystroke is disabled upon power up, the *en_alpha* function can be used to enable it.

For 8000 and 8300, there are two types of behaviors:

ALPHA_FIXED: only alphabet input is enabled, numeral input is disabled.

ALPHA_ROLLING: both alphabet and numeral inputs are enabled. For example, the “2ABC” key can generate ‘A’, ‘B’, ‘C’ and ‘2’, but if set to ALPHA_FIXED, only ‘A’, ‘B’ and ‘C’ can be generated.

return none

en_shift	**720**

purpose Enable the processing of shift keystroke.

syntax void en_shift (void);

example call en_shift ( );

description The *en_shift* function enables the shift key stroke processing. It is disabled upon power on.

return none

get_alpha_enable_state

purpose Check the status of alphabet keystroke processing.

syntax int get_alpha_enable_state (void);

example call state = get_alpha_enable_state ( );

description This routine gets the current status, enable or disable, of the alphabet key stroke processing. The default is enabled.

return 1, if the alphabet key stroke processing is enabled.
0, if disabled.

get_alpha_lock_state

purpose Get information of the alpha lock status.

syntax int get_alpha_lock_state (void);

example call state = get_alpha_lock_state( );

description This function returns an integer indicates the alpha lock status. The default is unlocked.

return 1, if alpha key is locked.
0, if alpha key is not locked.

getchar

purpose Get one character from the keyboard buffer.

syntax char getchar (void);

example call
```
c = getchar( );
if (c >0 ) printf ("Key %d pressed", c);
else printf ("No key pressed");
```

description The *getchar* function reads one key stroke from the keyboard buffer and then removes the key stroke from the keyboard buffer.

return The *getchar* function returns the character read from the keyboard buffer. If the keyboard buffer is empty, a null character (0x00) is returned.

get_shift_enable_state **720**

purpose Get the status of the shift keystroke processing.

syntax int get_shift_enable_state (void);

example call state = get_shift_enable_state( );

description This routine gets the current status, enable/disable, of the shift key stroke processing. The default is enabled.

return 1, if the shift key stroke processing is enabled.
0, if disabled.

get_shift_lock_state **720**

purpose Get shift lock state information.

syntax int get_shift_lock_state (void);

example call state = get_shift_lock_state( );

description This function returns an integer indicates the shift lock status. The default is unlocked.

return 1, if shift key is locked.
0, if shift key is not locked.

GetKeyClick

purpose Get the setting of the key click.

syntax int GetKeyClick (void);

example call state = GetKeyClick ( );

description By default, the key click is enabled, but the user can change this setting from within the system menu or through programming. This function allows you to retrieve the current setting of the key click.

return 0, if key click is disabled.
1, if key click is enabled. /* 711, 720, 8100 */
1~5, represents the tone of a key click /* 8000, 8300 */

kbhit

purpose Check if there is any key being pressed.

syntax int kbhit (void);

example call
```
for ( ;!kbhit( ); );    /* wait till key pressed */
```

description The *kbhit* function checks if there is any character waiting to be read from the keyboard buffer.

return If the keyboard buffer is empty, the *kbhit* function returns an integer value of 0. If the user presses a key and there is a character in the keyboard buffer, it returns 1.

peek_kb **711 720 8100**

purpose To detect multiple key-pressed combination from the keypad.

syntax unsigned long peek_kb (void);

example call
```
unsigned long keycode;
while (1)
    {
    keycode = peek_kb( );
    if ((keycode & SC_1) && (keycode & SC_2) && (keycode & SC_3))
        printf ("The user presses 1, 2, 3 simultaneously");
    OSTimeDly (8);              // delay 8x5 = 40ms
    }
```

description The *peek_kb* function scans the keyboard and returns an unsigned long integer to show the keys that are being pressed at the same time. The scan codes of these being-pressed keys are bit-wise OR together as the return value. Usually this is used to detect special key combinations for a special purpose. Note it may need up to 40ms for the system to scan the whole keyboard, therefore two consecutive calls should not be made within 40ms. A simple way to solve this problem is call the OSTimeDly function after the first call.

return An unsigned long integer is returned with the scan codes of the being-pressed keys are bit-wise OR together.

set_alpha_lock

purpose Set alpha lock state.

syntax void set_alpha_lock (int state);

int state; /* alpha lock state to be set , 1/0 to turn on/off */

example call set_alpha_lock (1); /* on alpha lock */

description This routine turns on or off the alpha lock.

return none

set_shift_lock	**720**

purpose Set shift lock state.

syntax void set_shift_lock (int state);
int state; /* shift lock state to be set , 1/0 to turn on/off */

example call set_shift_lock (1); /* on shift lock */

description This routine turns on or off the shift lock.

return none

SetKeyClick

purpose To enable / disable the key click.

syntax void SetKeyClick (int status);
int status; /* 0: disable key click, 1 ~ 5: enable key click */

example call SetKeyClick (1); /* enable key click sound */

description For 711, 720 and 8100, this routine turns on or off the key click. For 8000 and 8300, there are 5 different tones available. You can choose one of the tones from within the system menu or through programming. Besides, the key click's frequency and duration pair will be held in the system's global variable *KEY_CLICK* so that you can use it to generate the sound same as the key click. For example,
on_beeper (KEY_CLICK);

return none

shift_arrow	**720**

purpose To enable key combination of the shift and arrow keys.

syntax void shift_arrow (int state);
int state; /* 1/0 to enable/disable */

example call shift_arrow (1); /* enable shift+arrow key combination*/

description This routine can be used to enable the shift+arrow key combination, i.e., if the user hits the arrow key when the shift key state is on, it will generate KEY_SLEFT, instead of KEY_LEFT, etc.

return none

shift_func	**720**

purpose To enable key combination of the shift and function keys.

syntax int shift_func(int state);
int state; /* 1/0 to enable/disable */

example call	shift_func(1); /* enable shift+function key combination*/
description	This routine can be used to enable the shift+function key combination, i.e., if the user hits the function key when the shift key state is on, it will generate another function, instead of the original function.
return	none

TriggerStatus

purpose	To check if the scan key has been pressed.
syntax	int TriggerStatus (void);
example call	if (TriggerStatus( )) printf ("Scan key is pressed");
description	This function is used to check if the scan key has been pressed or not.
return	If scan key is pressed, it returns 1, else it returns 0.

2.9 LCD

The LCDs of the portable terminals are FSTN graphic display. For 711, 720, 8100 and 8300, the display size is 128x64 dots. As for 8000, it's 100x64 dots.

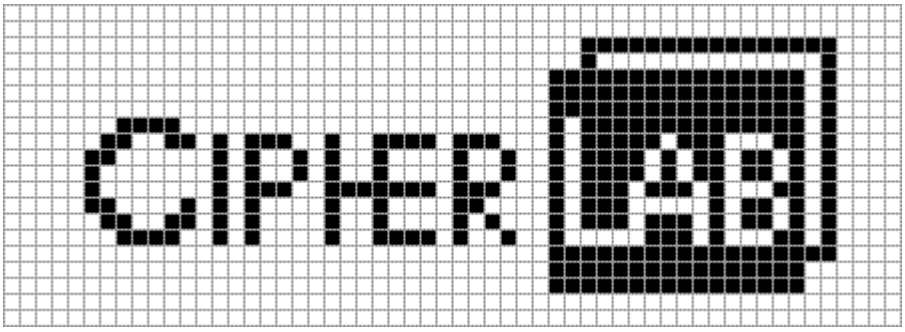
2.9.1 Graphic Display

All portable series support two different sizes of fonts, that is, 6x8 and 8x16. If 6x8 font is used, there are 8 lines and 20 characters per line at most. If 8x16 font is used, there are 4 lines and 15 characters per line at most. Different fonts can co-exist on the display, the user can call the "SetFont" function to change the font size whenever it's needed.

A coordinate system is used for the cursor movement routines to determine the cursor's location. The coordinate of the top left position is (0,0) and the bottom left position is assigned with (0,7). That is, for row positions, it's always from 0 to 7 (each row occupies 8 dots) regardless of the font size being used. As for column positions, it depends on the size of the font being used. For example, if an 8X16 font is used, the bottom right position will be (14,7).

For some graphics routines such as *clr_rect*, *fill_rect*, *show_image*, and *get_image*, the coordinate system used is on dot (pixel) basis. The top left position is (0,0), and the bottom right position is (99, 63) for 8000 and (127,63) for other portables.

The *show_image* function can be used to display images on the LCD. The user needs to allocate an unsigned char array to store the bitmap data of the image. This array begins with the top row of pixels. Each row begins with the left-most pixels. Each bit of the bitmap represents a single pixel of the image. If the bit is set to 1, the pixel is marked, and if it is 0, the pixel is unmarked. The 1st pixel in each row is represented by the least significant bit of the 1st byte in each row. If the image is wider than 8 pixels, the 9th pixel in each row is represented by the least significant bit of the 2nd byte in each row. The following is an example for showing the company logo and the static unsigned char array for storing its bitmap data.



static unsigned char CipherLab_logo[] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xf0, 0xff, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x10, 0x00, 0x08, 0x00, 0x00, 0x00, 0x00, 0xfc, 0xff, 0x0b, 0x00, 0x00, 0x00, 0x00, 0xfc, 0xff, 0x0b, 0x00, 0x00, 0x00, 0x00, 0xfc, 0xff, 0x0b, 0x80, 0x07, 0x00, 0x00, 0xf4, 0xff, 0x0b, 0xc0, 0xac, 0x93, 0x77, 0xf4, 0x1d, 0x0b, 0x60, 0xa0, 0x94, 0x90, 0xf4, 0xda, 0x0a, 0x20, 0xa0, 0x94, 0x90, 0xf4, 0xda, 0x0a, 0x20, 0xa0, 0xf3, 0x77, 0x74, 0x17, 0x0b, 0x60, 0xa8, 0x90, 0x30, 0x74, 0xd0, 0x0a, 0xc0, 0xac, 0x90, 0x50, 0x74, 0xd7, 0x0a, 0x80, 0xa7, 0x90, 0x97, 0x04, 0x17, 0x0b, 0x00, 0x00, 0x00, 0x00, 0xfc, 0xff, 0x0f, 0x00, 0x00, 0x00, 0x00, 0xfc, 0xff, 0x03, 0x00, 0x00, 0x00, 0x00, 0xfc, 0xff, 0x03, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

2.9.2 Font Files

Besides standard font, the terminal can display special characters provided that the font file for that character has been downloaded to the terminal. For 711, 720 and 8100, following font files are available:

- Font-jp.shx : Japanese kanji font
- Font-kr.shx : Korean font
- Font-sc.shx : Simplified Chinese font (size: 16x16 dots)
- Font-sd.shx : Simplified Chinese font (size: 12x12 dots)
- Font-tc.shx : Traditional Chinese font
- He-f6x8.rel : Hebrew font - C object file (size: 6x8 dots)
- He-f8x16.rel : Hebrew font - C object file (size: 8x16 dots)
- Po-f6x8.rel : Polish font - C object file (size: 6x8 dots)
- Ru-f6x8.rel : Russian font - C object file (size: 6x8 dots)
- Ru-f8x16.rel : Russian font - C object file (size: 8x16 dots)

For font files (FONT-XX.SHX), they need to be downloaded to the terminal. For “C” object files (*.REL), they need to be included inside the link file for linking with the “C” library and other “C” object files. Also, for showing Chinese, Japanese or Korean characters, the specific library needs to be included if the related functions are called in the “C” program. The specific libraries are listed as follows:

- 711/720/8100jplib.lib : including *jpprintf*, *jpputchar*, and *jpputs* functions
- 711/720/8100krlib.lib : including *krprintf*, *krputchar*, and *krputs* functions
- 711/720/8100sclib.lib : including *scprintf*, *scputchar*, and *scputs* functions
- 711/720/8100sdlib.lib : including *sdprintf*, *sdputchar*, and *sdputs* functions
- 711/720/8100tclib.lib : including *tcprintf*, *tcputchar*, and *tcputs* functions

For 8000 and 8300, following font files are available:

- 8xxx-He.shx : Hebrew font
- 8xxx-Jp.shx : Japanese kanji font
- 8xxx-kr.shx : Korean font
- 8xxx-Nd.shx : Nordic font
- 8xxx-Po.shx : Polish font
- 8xxx-Ru.shx : Russian font
- 8xxx-Sc.shx : Simplified Chinese font (size: 16x16 dots)
- 8xxx-Sd.shx : Simplified Chinese font (size: 12x12 dots)
- 8xxx-Tc.shx : Traditional Chinese font
- Multi-language.shx : multilingual fonts, including English (default), French, Hebrew, Latin, Nordic, Portugal, Russian, Slavic, Polish, Turkish and Slovak. Besides English, for showing any of them, you need to call the “*SetLanguage*” function to specify the font.

CheckFont	**8000**	**8300**

purpose	Check the current system font file.
syntax	int CheckFont (void);
example call	n = CheckFont ();
description	This function check the current font file that resides in the flash memory.
return	It returns one of the following integer numbers:

0	No additional font file, only the system default font is available
1	Traditional Chinese font (Big-5 code, 16x16 dots per char)
2	Simplified Chinese font (GB code, 12x12 dots per char)
3	Simplified Chinese font (GB code, 16x16 dots per char)
4	Korean font
5	Japanese font
6	Hebrew font
7	Polish font
8	Russian font
16	Multi-language font

clr_eol

purpose Clear from where the cursor is to the end of the line.

syntax void clr_eol (void);

example call clr_eol( );

description The *clr_eol* function clears from where the cursor is to the end of the line, and then moves the cursor to the original place.

return none

clr_icon

purpose Clear the icon zone area on the LCD display.

syntax void clr_icon (void);

example call clr_icon ();

description The icon zone is the unprintable area reserved for showing status icon such as battery icon. Users can also show their own icons on this area by using the "*show_image*" function. For 128x64 dots display, the icon zone is the right-most 8x64 dots. For 100x64 dots display, the icon zone is the right-most 4x64 dots. When calling the "*clr_scr()*" to clear the screen, this icon zone won't be cleared. Therefore if you need to erase the icon zone area, you have to call the "*clr_icon()*" function.

return none

clr_rect

purpose Clear a rectangular area on the LCD display.

syntax
```
void clr_rect (int left, int top, int width, int height);
int left;   /* x coordinate of the upper left corner of the rectangle */
int top;    /* y coordinate of the upper left corner of the rectangle */
int width;  /* the width in dots of the rectangle to be cleared */
int height;/* the height in dots of the rectangle to be cleared */
```

example call clr_rect (12, 8, 40, 8);

description The *clr_rect* function clears an rectangular area on the LCD display whose top left position and size are specified by *left, top, width,* and *height*. The cursor position is not affected after the operation.

return none

clr_scr

purpose	Clear LCD display.
syntax	void clr_scr (void);
example call	clr_scr( );
description	The *clr_scr* function clears the LCD display and places the cursor at the first column of the first line, that is (0,0) as expressed with the coordinate system.
return	none

DecContrast

purpose	Decrease the LCD contrast
syntax	void DecContrast (void);
example call	DecContrast();
description	The *DecContrast* function will decrease the LCD contrast by one level whenever it is being called. However, the lowest contrast is 0.
return	none.
See also	IncContrast, SetContrast.

fill_rect

purpose	Fill a rectangular area on the LCD display.
syntax	void fill_rect (int left, int top, int width, int height); int left; /* x coordinate of the upper left corner of the rectangle */ int top; /* y coordinate of the upper left corner of the rectangle */ int width; /* the width in dots of the rectangle to be filled */ int height; /* the height in dots of the rectangle to be filled */
example call	fill_rect (12, 8, 40, 8);
description	The *fill_rect* function fills a rectangular area on the LCD display whose top left position and size are specified by *left, top, width,* and *height*. The cursor position is not affected after the operation.
return	none

GetContrast **8000 8300**

purpose	To get the contrast level of the LCD
syntax	int GetContrast (void);
example call	int nContrastLevel = GetContrast ( );
description	The *GetContrast* function is used to get the contrast level of the LCD.
return	The current contrast level of the LCD, which ranging from 0 to 7.
See also	SetContrast, IncContrast, DecContrast.

GetCursor

purpose	Get current cursor status.

syntax int GetCursor (void);

example call if (GetCursor( ) == 0) puts ("Cursor Off");

description The *GetCursor* function check if the cursor is visible or not.

return The *GetCursor* function returns an integer of 1 if the cursor is visible (turned on), 0 if not.

GetFont

purpose Get current font information.

syntax int GetFont (void);

example call if (GetFont( ) == FONT8X16) puts ("Font : 8X16");

description The *GetFont* function returns the information about the current font type.

return The return value depends on the current font being used.

FONT6X8 : if 6X8 font is used
FONT8X16 : if 8X16 font is used

get_image

purpose Read the bitmap pattern of a rectangular area on the LCD display.

syntax
```
void get_image(int left, int top, int width, int height, unsigned char *pat);
int left;        /* x coordinate of the upper left corner of the rectangle */
int top;         /* y coordinate of the upper left corner of the rectangle */
int width;       /* the width in dots of the rectangle */
int height;      /* the height in dots of the rectangle */
unsigned char *pat; /* the buffer where bitmap data will be copied to */
```

example call get_image(12, 32, 60, 16, buf);

description The *get_image* function copies the bitmap pattern of a rectangular area on the LCD display whose top left position and size are specified by *left, top, width,* and *height* to the buffer specified by *pat*. The cursor position is not affected after the operation.

return none

GetVideoMode

purpose Get current display mode information.

syntax int GetVideoMode (void);

example call if (GetVideoMode( ) == VIDEO_NORMAL) puts("Normal Mode");

description The *GetVideoMode* function returns the information about the display mode.

return The return value depends on the current display mode being used.

VIDEO_NORMAL : if normal mode is selected
VIDEO_REVERSE : if reverse mode is selected

gotoxy

purpose Move cursor to new position.

syntax int gotoxy (int x_position, int y_position);
int x_position; /* x coordinate of the new cursor position desired */
int y_position; /* y coordinate of the new cursor position desired */

example call gotoxy (10, 0); /* move to the 11th column of the first line */

description The *gotoxy* function moves the cursor to a new position whose coordinate is specified in the argument *x_position* and *y_position*.

return Normally the *gotoxy* function will return an integer value of 1 when operation completes. In case of LCD fault, 0 is returned to indicate error.

ICON_ZONE

purpose Enable or disable the printing on the icon area.

syntax void ICON_ZONE (int mode);

int mode; /* 1: allowed to print, 0: not allowed (default) */

example call ICON_ZONE (1);

description The icon zone is the area reserved for showing status icon such as battery icon. By default, the icon zone is unprintable and can be accessed through graphic commands only. For 128x64 dots display, the icon zone is the right-most 8x64 dots. If it's enabled for printing, it can have 21 characters of small font per line and 16 characters of large font per line. But please note that the system may still show the battery and alpha icons in this icon area even it is set to printable for the user program. For 100x64 dots display, the icon zone is the right-most 4x64 dots. Since 4 pixels width cannot hold one character, the characters per line remain the same even it's set to printable. But for both 128x64 and 100x64 dots display, if the icon zone is set to printable, after calling the "*clr_scr()*" function, the entire screen will be erased.

return none

IncContrast

purpose Increase the LCD contrast

syntax void IncContrast (void);

example call IncContrast();

description The *IncContrast* function will increase the LCD contrast by one level whenever it is being called. However, the highest contrast level is 7.

return none.

See also IncContrast, SetContrast.

lcd_backlit

purpose Set LCD backlight

syntax void lcd_backlit (int state);

int state; /* LCD backlight state 0 / 1 (off / on) */

example call lcd_backlit (1); /* turn on LCD backlight */

description The *lcd_backlit* turns the LCD backlight on or off depending on the value of *state*. The backlight will be on if *state* is 1, off if 0. The system global

variable **BKLIT_TIMEOUT** can be used to specify the backlight duration in unit of second. But if this value is set to zero, the backlight will be on until the backlight state is set to off or user turn off it manually.

return none.

printf

purpose Write character strings and values of C variables in a specified format to the LCD display.

syntax
```
int printf (char* format, var);
char* format;     /* character string that describes the format to be used
                     variable number of arguments whose values are being
                     printed on the LCD display */
```

example call
```
printf ("ID : %s", id_buffer);
```

description The *printf* function accepts a variable number of arguments and prints them to the LCD display. The value of each argument is formatted according to the codes embedded in the format specification *format*.

To print values of C variables, a format specification must be embedded in *format* for each variable to be printed. The format specification for each variable has the following form :

%[flags][width].[precision][size][type]

Field	Explanation
% (required)	Indicates the beginning of a format specification. Use %% to print a percentage sign.
flags (optional)	One or more of the '-', '+', '#' characters or a blank space specifies justification, and the appearance of plus / minus signs in the values printed (see table below).
width (optional)	A number that indicates how many characters, at a minimum, must be used to print the value
precision (optional)	A number that specifies how many characters, at maximum, can be used to print the value. When printing integer variables, this is the minimum number of digits used.
size (optional)	A character that modifies the type field which comes next. One of the characters 'h', 'l', 'L' can appears in this field to differentiate between short and long integers. 'h' is for short integers, and 'l' or 'L' for long integers.
type (required)	A letter that indicates the type of variable being printed (see table below)

Flags	Meaning
-	Left justify output value. Default is right justification.
+	If the output value is a numerical one, print a '+' or '-' character according to the sign of the value. A '-' character is always printed for a negative value no matter this flag is specified or not.
blank	Positive numerical values are prefixed with blank spaces. This flag is ignored if the + flag also appears.
#	When used in printing variables of type o, x, or X, none zero output values are prefixed with 0, 0x, or 0X, respectively.

Type	Expected Input
c	Single character.

d	Signed decimal integer.
i	Signed decimal integer.
o	Octal digits without sign.
u	Unsigned decimal integer.
x	Hexadecimal digits using lower case letter.
X	Hexadecimal digits using upper case letter.
s	A null terminated character string.

The *jpprint*, *scprintf*, and *tcprintf* functions are special *printf* functions to display a string that consists of the Japanese, simplified Chinese and/ or traditional Chinese characters and the other variables.

return The *printf* function returns the number characters sent to the LCD display

putchar

purpose Display a character on the LCD display.

syntax
```
int putchar (char c);
char c;      character sent to the LCD display
```

example call
```
putchar('A');
```

description The *putchar* function sends the character specified in the argument *c* to the LCD display at the current cursor position and moves the cursor accordingly.

The *jpputchar*, *scputchar*, and *tcputchar* functions are special *putchar* functions to display a single Japanese, simplified Chinese and/ or traditional Chinese character.

return none

puts

purpose Display a string on the LCD display.

syntax
```
char puts (char* string);
char* string;      /* string to be displayed */
```

example call
```
puts ("Password : ");
```

description The *puts* function sends a character string whose address is specified in the argument *string* to the LCD display starting from the current cursor position. The cursor is moved accordingly as each character of *string* is sent to the LCD display. The operation continues until a terminating null character is encountered.

The *jpputs*, *scputs*, and *tcputs* functions are special *puts* functions to display a string which consists of the Japanese, simplified Chinese and/ or traditional Chinese characters.

return The *puts* function returns the number characters sent to the LCD display

SetContrast

purpose To set contrast level for the LCD

syntax
```
void SetContrast (int level);
```

example call
```
SetContrast (4);
```

description The *SetContrast* function is used to set the contrast level for LCD. The valid level is ranging from 0 to 7, which the higher value has higher contrast.

return none.

See also GetContrast, IncContrast, DecContrast.

SetCursor

purpose Turn on or off the cursor of the LCD display.

syntax void SetCursor (int status);
int status; /* integer representing cursor status to be set */

example call SetCursor (0); /* invisible the cursor */

description The *SetCursor* function displays or hides the cursor of the LCD display according to the value of *status* specified. If *status* equals 1, the cursor will be turned on to show the current cursor position. If *status* equals 0, the cursor will be invisible.

return none.

SetFont

purpose Select what size of font to be used afterwards.

syntax void SetFont (int font);
int font; /* integer representing font to be used */

example call SetFont (FONT_8X16);

description This function is used to specify what size of font to be used following this call. The valid values are as follows

FONT_6X8: 6x8 graphic dots per character

FONT_8X16: 8x16 graphic dots per character

return none

SetLanguage **8000 8300**

purpose Select which language to be used from the multi-language font file.

syntax void SetLanguage (int setting);
int setting; /* integer represents the language to be used */

example call SetLanguage (NORDIC); /* choose the Nordic font */

description If the “Multi-language.shx” font file has been downloaded to the terminal, then this function can be used to specify which language font is to be used by the system. The valid values are as follows

0x10: the standard ASCII characters
0x11: French
0x12: Hebrew
0x13: Latin
0x14: Nordic
0x15: Portugal
0x16: Russian

0x17: Slavic
0x18: Polish
0x19: Turkish
0x1a: Slovak

return none

SetVideoMode

purpose Select video mode for the display.

syntax
```
void SetVideoMode (int mode);
int mode;        /* integer representing video mode to be set */
```

example call
```
SetVideoMode (VIDEO_REVERSE);   /* select reverse video mode */
```

description The *SetVideoMode* function set the display mode for the following LCD operation. The available modes are VIDEO_NORMAL and VIDEO_REVERSE.

return none

show_image

purpose Put a rectangular bitmap to the LCD display.

Syntax
```
void show_image (int left, int top, int width, int height, unsigned char
*pat);
int left;       /* x coordinate of the upper left corner of the rectangle */
int top;        /* y coordinate of the upper left corner of the rectangle */
int width;   /* the width in dots of the rectangle */
int height;  /* the height in dots of the rectangle */
unsigned char *pat; /* the buffer that hold the bitmap to be displayed */
```

example call
```
show_image (35, 5, 52, 24, CipherLab_logo[]);
```

description The *showet_image* function displays a rectangular bitmap specified by *pat* to the LCD display. The rectangular's top left position and size are specified by *left, top, width,* and *height*. The cursor position is not affected after the operation.

return none

wherex

purpose Get x-coordinate of the cursor location.

syntax
```
int wherex (void);
```

example call
```
x_position = wherex ( );
```

description The *wherex* function determines the current x-coordinate location of the cursor.

return The *wherex* function returns the x-coordinate of the cursor location.

wherexy

purpose Get x-coordinate and y-coordinate of the cursor location

syntax int wherexy (int* column, int* row);
int* column; /* pointer to integer where x-coordinate is stored */
int* row; /* pointer to integer where y-coordinate is stored */

example call wherexy (&x_position, &y_position);

description The *wherexy* function copies the value of x-coordinate and y-coordinate of the cursor location to the variables whose address is specified in the arguments *column* and *row*.

return none

wherey

purpose Get y-coordinate of the cursor location.

syntax int wherey (void);

example call y_position = wherey( );

description The *wherey* function determines the current y-coordinate location of the cursor.

return The *wherey* function returns the y-coordinate of the cursor location.

2.10 Power

All portable terminals have a main battery for normal operation and a backup battery for keeping SRAM data as well as time accuracy. This section describes the power management functions that can be used to monitor the voltage levels of the main battery and backup battery.

charger_status **8300**

purpose Get the status of the battery charging.

syntax int charger_status (void);

example call
```
if (charger_status == CHARGE_DONE)
    puts (“Battery is full”);
```

description This function checks the charging conditions for the main battery.

return One of the following constants,

CHARGE_STANDBY -- 0x00, not connected to any external power
CHARGING -- 0x01, the battery is being charged now
CHARGE_DONE -- 0x02, the battery is full
CHARGE_FAIL -- 0x03, unable to charge the battery

get_vmain

purpose Get voltage level of the main battery.

syntax unsigned get_vmain (void);

example call
```
if (get_vmain( ) < 2200)          // alkaline batteries
    puts (“Battery is low”);
```

description This function reads the voltage level of the main battery in units of mV.

return The voltage level of the main battery in units of mV (milli-volt).

get_vbackup

purpose Get voltage level of the backup battery.

syntax unsigned get_vbackup (void);

example call
```
bat1 = get_vbackup( );
```

description This function reads the voltage level of the backup battery in units of mV.

return The voltage level of the backup battery in units of mV (milli-volt).

2.11 Communication Ports

There are two communication ports on each terminal, namely COM1 and COM2. The program needs to call the "**SetCommType**" function to set up the communication type for the COM ports before using them. The following table shows the mappings of the communication ports for the portable series.

	COM1	COM2
711	RS-232	Serial IR, IrDA
720	RS-232, RS-485	Serial IR, IrDA
8000	Serial IR, IrDA	RF
8100	RS-232	RF
8300	RS-232, Serial IR, IrDA	RF

Besides the data signals (transmit & receive), 2 handshake signals (RTS & CTS) are also provided for flow control. Features provided are described in detail below,

2.11.1 Parameters

- Baud rate : One out of the 8 baud rates can be selected (115200, 76800, 57600, 38400, 19200, 9600, 4800, 2400)
- Data bits : 7 or 8
- Parity : Even, Odd or none
- Stop bit : 1
- Flow control : RTS/CTS, XON/XOFF, or None

2.11.2 Receive Buffer

A 256 bytes FIFO buffer is allocated for each port. The data successfully received is stored in this buffer sequentially (if any error such as framing, parity error and so on occurs, the data is simply discarded). However if the buffer is full, the data followed will be discarded and an overrun flag is set to indicate this error.

2.11.3 Transmit Buffer

The system does not allocate any transmit buffer, it simply records the pointer of the string to be sent. The transmission stops when a null (0x00) character is encountered. The application program must allocate its own transmit buffer and not to modify it during transmission.

2.11.4 Flow Control

To avoid data loss, 3 kinds of flow control are supported and are done by background routines.

1) None : no flow control is performed

2) CTS : RTS and CTS signals are used for flow control.

    - Transmit : The transmission is allowed only when CTS signal is at the active level (mark). If the CTS is dropped and later become active again, the transmission is automatically resumed by background routines. However, due to the UART design (on-chip temporary transmission buffer), up to 2 characters might be sent after the CTS was dropped.

- Receive : The RTS signal is used to indicate that the receiving buffer is or is going to be full and instruct the transmitting side to halt the transmission. If there are less than 5 character spaces available in the receiving buffer, the RTS is dropped. Then the RTS is activated again when there are no less than 10 character spaces available in the receiving buffer. If there are sufficient spaces in the buffer, the received data is stored even when RTS is dropped.

3) XON/XOFF : instead of RTS/CTS signals, 2 special characters are used for flow control. That is, XON (hex 11) and XOFF (hex 13). XON is used to enable transmission while XOFF to disable transmission.

- Transmit : when the port is opened, the transmission is enabled. Then every character received is examined to see if it is a normal data or flow control codes. If XOFF is received, transmission is halted. It is resumed later when a XON is received. Just like RTS/CTS control, up to 2 characters might be sent after the XOFF was received.

- Receive : The received characters are examined to see if it is normal data (stored into receive buffer) or flow control codes (set/reset transmission flag but not stored). If there are less than 5 character spaces available in the receiving buffer, the XOFF is sent. Then the XON is sent when there are no less than 10 character spaces available in the receiving buffer. If there are sufficient spaces in the buffer, the received data is stored even when in XOFF state. **Note** that if receiving/transmission are concurrently in operation, XON/XOFF control codes might be inserted into normal transmit data string. In using this method, make sure the respective side features the same control methodology or dead lock might happen.

Regardless which flow control method is selected, the RTS is activated when the port is *opened* and dropped when the port is *closed* (default).

clear_com

purpose Clear the data receiving buffer

syntax
```
void clear_com (int port);
int port;        /* the port number (1 or 2) of the receiving buffer */
```

example call
```
clear_com(1);    /* clear COM1's receiving buffer */
```

description This routine is used to clear all the data stored in the receiving buffer. This can be used to avoid mis-interpretation when overrun or other error occurred.

return none

close_com

purpose To close the specified communication port

syntax
```
void close_com (int port);
int port;        /* port to be closed, either 1 or 2 */
```

example call
```
close_com(1);    /* close com1 */
```

description The *close_com* disables the communication port specified.

return none

com_cts

purpose Get CTS level

syntax int com_cts (int port);
int port; /* the port number, either 1 or 2 */

example call if (com_cts(1) == 0) printf ("COM1 CTS is space");
else printf("COM1 CTS is mark");

description This routine is used to check current CTS level.

return 1, if CTS is in mark state
0, if CTS is in space state

com_eot

purpose To see if any COM port transmission in process (End Of Transmission)

syntax int com_eot (int port);
int port; /* the port number, either 1 or 2 */

example call while (!com_eot(1)); /* wait till prior transmission completed */
write_com (1, "NEXT STRING");

description This routine is used to check if prior transmission is still in process or not.

return 0, prior transmission is still in course
1, transmission is completed

com_overrun

purpose See if overrun error occurred

syntax int com_overrun (int port);
int port; /* the port number, 1 or 2 */

example call if (overrun(1) > 0) clear_com(1);
/* if overrun, data stored in the buffer is not complete, clear them */

description This routine is used to see if overrun met. The overrun flag is automatically cleared after examined.

return 1, overrun error met
0, OK

com_rts

purpose Set RTS signal

syntax void com_rts (int port, int i);
int port; /* the port number, either 1 or 2 */
int i; /* RTS state, 1/0, mark/space */

example call com_rts (1, 1);/* set COM1 RTS to mark */

description This routine is used to control the RTS signal. It works even when the CTS flow control is selected. However, RTS might be changed by the background routine according to receiving buffer status. It is strongly recommended not to use this routine if CTS control is utilized.

return none

nwrite_com

purpose Send a specific number of characters out through the COM port

syntax void nwrite_com (int port, char *s, int count);
int port; /* the port number, either 1 or 2 */
char *s; /* string to be sent */
int count; /* number of character to be sent */

example call char s[ ] = { "Hello\n" };
nwrite_com (1, s, 2); /* send two characters "He" through COM1 */

description This routine is used to send a specific number of characters specified by *count* through COM ports. The character string is transmitted one by one until the specified number of character is sent.

return none

open_com

purpose Initialize and enable the specified COM port

syntax void open_com (int port, int parameter);
int port; /* port to be opened, either 1 or 2 */
int parameter; /* port parameters as below */

D0-D2	baud rate	0 to 7 = 115200/76800/57600/ 38400/19200/9600/4800/2400
D3	data bits	0 : 7bits 1 : 8 bits
D4	Parity enable	0 : disable 1 : enable
D5	even/odd	0 : odd 1 : even
D6	flow control	0 : disable 1 : enable
D7	flow control method	0 : CTS, 1 : XON/XOFF

example call open_com (1, 0x0b);
/* open com1 to 38400, 8 data bits, no parity and no handshake */

description The open_com function initializes the specified COM port, clears its receiving buffer, stops any on going data transmission, reset COM port status and configure the COM port according to the setting.

return none

read_com

purpose Read 1 byte from the COM port receiving buffer

syntax int read_com (int port, char *c);
int port; /* the port number, either 1 or 2 */
char *c; /* pointer to character returned */

example call
```
char c;
if (read_com (1, c))
   printf ("char %c received from COM1", *c);
```

description This routine is used to read one byte from the receive buffer and then remove it from the buffer. However, if the buffer is empty, no action is taken and 0 is returned.

return 1, available or 0 if buffer is empty

SetCommType

purpose Set communication type for the port specified.

syntax
```
int SetCommType (int port, int type);
int port;   /* port to be set, can be either 1 or 2 */
int type;   /* communication types, following types are available */
```
0: direct RS-232

1: docking (via communication cradle)

2: Serial IR (via IR transceiver)

3: standard IrDA communication

4: RF communication

Note a COM port can support only some of the communication types, please refer to the following table for the COM port mappings of each terminal:

	COM1	COM2
711	RS-232	Serial IR, IrDA
720	RS-232, RS-485	Serial IR, IrDA
8000	Serial IR, IrDA	RF
8100	RS-232	RF
8300	RS-232, Serial IR, IrDA	RF

example call
```
SetCommType (1, 2); /* set COM1 (8000/8300) to IR communication */
```

description This routine is used to set the communication types for the COM ports. Before opening the COM port, please call this function to assign communication type for the port, otherwise the communication may not work properly.

return 1 for valid setting (successful), 0 for invalid setting (failed).

write_com

purpose Send a null-terminated string out through the COM port

syntax
```
void write_com (int port, char *s);
int port;       /* the port number, either 1 or 2 */
char *s;        /* string to be sent */
```

example call
```
char s[ ] = { "Hello\n" };
write_com (1, s);           /* send String "Hello\n" through COM1 */
```

description This routine is used to send a string through the COM port. If any prior transmission is still in process, it is terminated then the current transmission resumes. The character string is transmitted one by one until a NULL character is met. A null string can be used to terminate prior transmission.

return none

2.12 RF Communication

The 8110 and 8310 terminals are equipped with 433MHz RF modules, and the 8150 and 8350 terminals are equipped with 2.4GHz RF modules.

2.12.1 RF Specifications

The specifications of the RF modules are as follows:

433 MHz RF module Specifications

- Frequency range: 433.12 ~ 434.62 MHz
- Data rate: 9600 bps
- Programmable channels: 4
- Coverage: 200M line-of-sight
- Maximum output power: 10mW (10dbm)
- Modulation: FSK (Frequency Shift Keying)
- Compliance: CE and FCC

2.4 GHz RF module Specifications

- Frequency range: 2.4000 ~ 2.4835 GHz, unlicensed ISM Band
- Type: Frequency Hopping Spread Spectrum Transceiver
- Data rate: 19200 bps
- Programmable channels: 6
- Coverage: 1000M line-of-sight
- Maximum output power: 100mW
- Compliance: CE and FCC

2.12.2 IDs and Groups

An ID to a terminal / base is just like a name to a person. Each terminal / base in the same RF system should have a unique ID. If the IDs are duplicated, the system may not work properly. So before running your RF system, please make sure that every terminal / base has a unique ID.

For our 433MHz RF system, up to 45 terminals and 16 bases can be supported by one system. The valid ID ranges from 1 to 45 for terminals, and 1 to 16 for bases. To support all 45 terminals, the 433MHz RF bases need to be configured to 3 groups. Each group and also each base can support up to 15 terminals.

- Base IDs (433MHz): 01 ~ 16
- Terminal IDs (433MHz): 01 ~ 45 (3 groups)
  01 ~ 15: supported by Group #1 Bases
  16 ~ 30: supported by Group #2 Bases
  31 ~ 45: supported by Group #3 Bases

For 2.4GHz RF system, up to 99 terminals and 16 bases can be supported by one system, and they all belong to the one group.

- Base IDs (2.4GHz): 01 ~ 16
- Terminal IDs (2.4GHz): 01 ~ 99

2.12.3 RF Bases

The RF terminals must communicate with the RF base stations. There are also two types of RF bases: the 433MHz RF base (3510) and 2.4GHz RF base (3550). The connection from the host computer to the base is RS-232, while the connection between bases is RS-485. Up to 16 bases can be connected together in one RF system. If two or more bases are connected together, the one connected to the host computer should be set to master mode, and the others in slave mode. Detailed specifications are as follows:

433 MHz Base Properties

- Mode: 1-standalone, 2-slave, 3-master
- Channel: 1 ~ 4
- ID: 01 ~ 16
- Group: 1 ~ 3
- Time out: 1 ~ 99 seconds, duration of retries for sending data
- Output power: 1~5 levels (10, 5, 4, 0, -5dBm)
- RF data rate: 9600 bps
- RS-232 baud rate: 115200, 57600, 38400, 19200, 9600

2.4 GHz Base Properties

- Mode: 1-standalone, 2-slave, 3-master
- Channel: 1 ~ 6
- ID: 01 ~ 16
- Group: 1
- Time out: 1 ~ 99 seconds, duration of retries for sending data
- Output power: maximum 100mW
- RF data rate: 19200 bps
- RS-232 baud rate: 115200, 57600, 38400, 19200, 9600

2.12.4 Terminal properties

The properties of the two kinds of RF terminals are as follows:

433 MHz RF Terminal

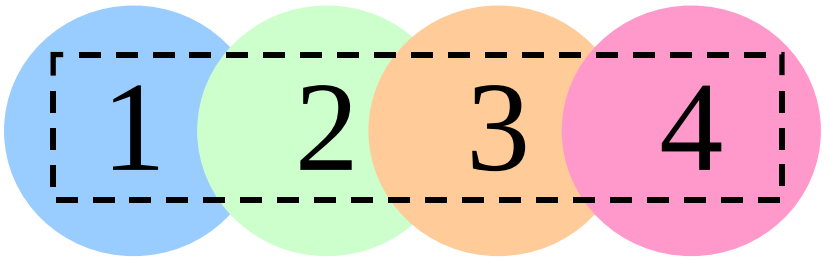
- Channel: 1 ~ 4
- ID: 01 ~ 45
- Time out: 1 ~ 99 seconds, duration of retries for sending data
- Output power: 1~5 levels (10, 5, 4, 0, -5dBm)
- Auto search: 0 ~ 99 sec, automatically search for available channels when connection to current channel is lost

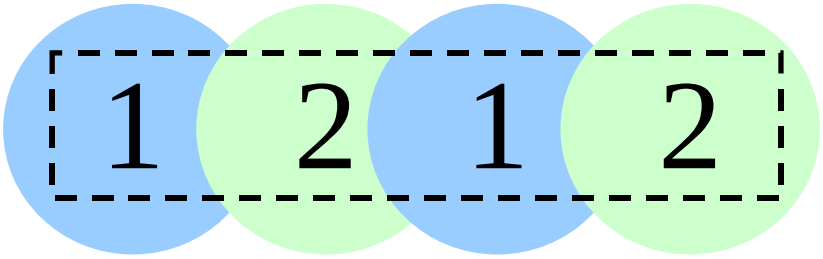
2.4 GHz RF Terminal

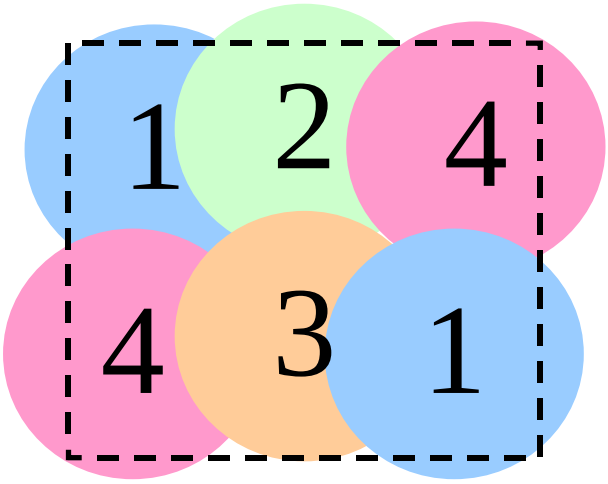
- Channel: 1 ~ 6
- ID: 01 ~ 99
- Time out: 1 ~ 99 seconds, duration of retries for sending data
- Auto search: 0 ~ 99 sec, automatically search for available channels when connection to current channel is lost

2.12.5 RF Topology & Roaming

In order to cover the whole working environment, you may need to install more than one base. The rule is simple: **two adjacent bases must not use the same channel**. Follows are some possible deployments for the 433MHz RF system. Each circle represents the area covered by one base, and the number is the base's channel number.







By default, a terminal can automatically switch to a different channel depending on its location. This is called **Roaming**. Unless the terminal is set to a fixed channel, whenever it can not find the base that it registers to, it will change its channel and try to connect to another base inside the system. With this auto-switching capability, the virtual working area for one terminal is not restricted to the coverage of one base, but the union area of the whole system.

2.12.6 RF Sysetm Deployment

Before deploying an RF system, some tasks such as site surveying, planning, installation and testing should be performed accordingly. Following are the guidelines for site surveying and planning:

- **Test the base station's coverage.**
  The coverage may be affected by environmental factors such as:
  - Background noise from other equipment.
  - Interference from other nearby RF systems.
  - Shielding by metallic or concrete structures of the building.
  - Humidity and other wave absorption materials.
- **Estimate the number of base stations needed.**
- **Estimate the number of terminals needed.**
- **Determine the base station's deployment topology.**

If possible, always install the RF base into the center of the working environment to get a better coverage. And, install the base on a higher level so that the antenna is in as much as possible free space.

After installation, select the maximum output power (10dbm) to test the transmission range and response time. Then try out each channel to get the best performance. If the range is good enough, repeat above procedure with lower output power (433MHz RF only). Using lower output power not just conserve battery power, but also reduce the possibility of interference to other RF systems.

int SetRFID (int ID)

Purpose	Sets the terminal ID.
Parameter	ID: 1 ~ 45 for 433MHz RF, 1 ~ 99 for 2.4GHz RF
Return	the ID, if successful
	0, if unsuccessful
See also	GetRFID

int GetRFID (void)

Purpose	Retrieves the terminal ID.
Parameter	none
Return	the terminal's ID
See also	SetRFID

int SetRFChannel (int channel)

Purpose	Sets the channel for the terminal.
Parameter	channel: 1 ~ 4 for 433MHz RF, 1 ~ 6 for 2.4GHz RF
Return	channel, if successful
	0, if unsuccessful
See also	GetRFChannel

int GetRFChannel (void)

Purpose	Retrieves the channel for the terminal.
Parameter	none
Return	the channel number, 1 ~ 4 for 433MHz RF, 1 ~ 6 for 2.4GHz RF
See also	SetRFChannel

void SetRFTimeOut (int sec)

Purpose	Sets the duration of retries, in seconds, for sending data.
Parameter	the duration in seconds, if set to 0, it's determined by the system.
	default: 5 seconds
Return	none
See also	SearchRFChannel

int SearchRFChannel (int sec)

Purpose	To automatically search for available channel when connection to base is lost for the specified period.
Parameter	time in seconds, default to 10
	if set to 0, the auto search will be disable.
Return	the channel number, 1 ~ 4 for 433MHz RF, 1 ~ 6 for 2.4GHz RF
	0, if the base is not found
See also	SetRFChannel, SetRFTimeOut

int CheckRFBase (void)

Purpose	To check if the terminal is connected to a base.
Parameter	none
Return	0, no base found
	1, base present
See also	SearchRFChannel

int SetRFPower (char p)

Purpose	Sets the RF output power (for 433MHz RF only).
Parameter	the power level 1 ~ 5
	1: 10dbm, 2: 5dbm, 3: 4dbm, 4: 0dbm, 5: -5dbm
Return	the original power level
See also	GetRFPower

int GetRFPower (void)

Purpose	Gets the RF output power (for 433MHz RF only).
Parameter	none
Return	the original power level
	1: 10dbm, 2: 5dbm, 3: 4dbm, 4: 0dbm, 5: -5dbm
See also	SetRFPower

2.13 Memory

All portable terminals have 1MB Flash memory for program storage and up to 4MB SRAM for data storage. The flash memory can also be used to store crucial data such as application settings. The flash memory is divided into 16 banks (64MB each) and the system has reserved one bank which address starts at 0xF60000 for this special purpose. But because of the characteristics of the flash memory, it needs to be erased before writing. The benefit of writing data to flash memory is that even the backup battery is gone, the data is still there.

EraseSector

purpose	To erase a whole sector of flash memory.
syntax	int EraseSector (void* sector_start_addr);
example call	EraseSector (0xf60000);
description	Before calling *WriteFlash* to write data into flash memory, the flash memory should be erased first by calling this function.
return	Number of bytes that has been erased.

FlashSize

purpose	To check the size of flash memory
syntax	int FlashSize (void);
example call	FlashSize ( );
description	The *FlashSize* function allows you to check the flash memory that available for the user program.
return	Flash memory size in K bytes.

free_memory

purpose	Get free memory size information.
syntax	long free_memory (void);
example call	available_memory = free_memory ( );
description	The *free_memory* function gets the information of the amount of free (unused) memory of the file space.
return	The *free_memory* function returns a long integer indicating the amount of free memory in bytes.

init_free_memory

purpose	Initialize file space.
syntax	void init_free_memory (void);
example call	init_free_memory ( );
description	The *init_free_memory* function will first try to identify how many SRAMs are installed, and then initialize the contents of the file space (total SRAM installed excludes memory of system space and user space). The original contents of the file space will be wiped out after this function is called. Whenever the amount of the SRAM installed is changed, this function must be called to recognize the changes.

return	This function has no return values.

RamSize

purpose	To check the size of data memory (SRAM).
syntax	int RamSize (void);
example call	RamSize( );
description	The *RamSize* function allows you to check the SRAM size that could be used for storing data.
return	RAM size in K bytes

WriteFlash

purpose	To write data to the flash memory.
syntax	int WriteFlash (void *target_addr, void *source_addr, unsigned long size);
example call	char szData[100]; EraseSector (0xf60000); WriteFlash (0xf60000, szData, 100);
description	The flash memory can also be used to store data if it's not fully used by user program. The possible available flash memory is 64 Kbytes and its address starts from 0xF60000.
return	Number of bytes that has been written to flash.

2.14 Smart-Media Card (720 only)

The data memory of CPT-720 can be extended to 8 Mbytes with a Smart-Media card. Following are the routines for manipulating it. To use these routines, please include the "smc.h" in your "C" source files.

free_byte_cnt

purpose To get the number of unallocated bytes available on the Smart-Media card.

syntax unsigned long free_byte_cnt (FILE * file_pointer);

FILE * f_pointer; /* pointer to FILE data structure */

Example

```
FILE *fp;
unsigned long freebytes;
fp = fopen ("A:\\file1", "r+");      /* opened for read & write */
freebytes = free_byte_cnt (fp);
```

return The number of bytes available to be allocated on the Smart-Media card associated with the file is returned

clearerr

purpose To reset the error and end-of-file indicators of a file specified by a file pointer.

syntax void clearerr (FILE *file_pointer);

FILE *file_pointer; /* pointer to the FILE data structure associated with the file whose error flag is being cleared */

description The *clearerr* function sets to zero the end-of-file and error flag associated with the file pointed to by the file_pointer. This flag has a nonzero value after an error or an end-of-file condition occurs. The error indicator for the file remains set until cleared by calling clearerr. These conditions may be verified by calling *ferror* and *feof*, respectively.

example

```
FILE *fp;
char string [81];
if ((fp = fopen ("A:\\file1", "r")) == NULL)     {
    printf ("fopen failed.\n");
    exit (0);
    }
fgets (string, 80, fp);
if (ferror (fp) != 0)     {
    printf ("Error detected\n");
    clearerr (fp);
    printf ("Error cleared\n");
    }
```

return None.

see also ferror, feof

fclose

purpose To close a file opened earlier for buffered input and output using *fopen*.

syntax
```
int fclose (FILE  *file_pointer);
FILE *file_pointer;                /* pointer to file to be closed */
```

example
```
FILE *fp;
fp = fopen (“A:\\file1”, “r+”);      /* opened for read & write */
/* processing */
if ( fclose (fp) )
   printf (“file close error);
```

description The *fclose* function closes the file specified by the argument file_pointer. If the file is open for writing, the contents of the buffer associated with the file are flushed before the file is closed.

return This function returns zero if the file was successfully closed, or EOF if any errors were detected. The contents of the buffer associated with the file will be flushed before the file is closed

see also fopen

feof

purpose To determine whether the end of a file has been reached.

syntax
```
int fclose (FILE *file_pointer);
FILE *file_pointer;     /* pointer to the FILE data structure associated
                           with the file whose status is being checked */
```

example
```
FILE *fp;
int  c;
fp = fopen (“A:\\file1”, “r+”);      /* opened for read & write */
while ( !feof (fp) ) {
      c = fgetc (fp);
      }
```

return This function returns a non-zero value if the end of the file is reached.

ferror

purpose To determine if an error has occurred during a previous read or write operation on a file.

syntax
```
int ferror (FILE *file_pointer);
FILE *file_pointer;     /* pointer to the FILE data structure associated
                           with the file whose status is being checked */
```

example
```
FILE *fp;
int  c;
fp = fopen (“A:\\file1”, “r+”);      /* opened for read & write */
while ( !feof (fp) ) {
      c = fgetc (fp);
      if (ferror (fp)) {
            printf (“Error detected\n”);
```

```
        clearerr (fp);
        printf ("Error cleared\n");
        }
    }
```

return This function returns a non-zero value if an error has occurred during a read or a write operation. Otherwise, it returns a 0.

see also clearerr

fflush

purpose To flush the output buffer associated with a file opened for buffered I/O. This will cause any remaining data in the output buffer written to the file.

syntax

```
int fflush (FILE *file_pointer);
FILE *file_pointer;     /* pointer to the FILE data structure associated
                           with the file whose buffer is being flushed */
```

example

```
FILE *fp;
if (fflush (fp)) {
    /* file flush error */
    }
```

return If the buffer is successfully flushed, *fflush* returns a 0. In case of an error, the return value is the constant EOF defined in *stdio.h*.

fgetc

purpose To read a single character from a file opened for buffered input.

syntax

```
int fgetc (FILE *file_pointer);
FILE *file_pointer;     /* pointer to the FILE data structure associated with
                           the file from which a character is to be read */
```

description The *fgetc* function reads a character from the current position of the file pointed to by the file_pointer and then increments this position. The character is returned as an integer.

example

```
FILE *fp;
char buffer[81];
int  i, c;
if ((fp = fopen ("A:\\file1", "r")) == NULL)
    {
    printf ("fopen failed.\n");
    exit (0);
    }
c = fgetc (fp);
for (i=0; (i < 80) && (feof (fp) == 0) && (c != '\n'); i++)
    {
    buffer [i] = c;
    c = fgetc (fp);
    }
```

buffer [i] = '\0';

printf ("First line of A:file1 : %s\n", buffer);

return If there are no errors, *fgetc* returns the character read. Otherwiae, it returns the constant EOF. Call *ferror* and *feof* to determine if there was an error or the file simply reached its end.

see also fgets, fputc, fputs

fgetpos

purpose To get and save the current read or write position of a file.

syntax int fgetpos (FILE *file_pointer, unsigned long *position);

FILE *file_pointer; /* pointer to the FILE data structure associated with the file whose current position is requested */

unsigned long *position; /* pointer to location where file's current position is returned */

description The *fgetpos( )* function fills *position* with a value representing the current position of the file pointed to by the file pointer. This is usually the byte number from the beginning of the file. In the case of a file open in text mode this may not be the same as the actual number of bytes you have read from the file. The *position* returned by *fgetpos()* should be used as an argument to *fsetpos()* to reposition a file to a former location..

example

```
FILE *fp;
int   c;
unsigned long position;
if ((fp = fopen ("A:\\file1", "r")) == NULL)
    {
    printf ("fopen failed.\n");
    exit (0);
    }
c = fgetc (fp);
if (fgetpos (fp, &position) != 0)
    printf ("fgetpos failed");
```

return The *fgetpos* returns a zero when successful. In case of error, the return value is nonzero and the global variable *errno* is set to the constant EBADF if the *file_pointer* does not point to a file or if it points to an inaccessible file.

see also fsetpos

fgets

purpose To read a line from a file opened for buffered input. The line is read until a newline (\n) character is encountered or until the number of characters reaches the specified maximum.

syntax char *fgets (char *string, int max_char, FILE *file_pointer);

char *string; /* pointer to buffer where characters are stored */

int max_char; /* maximum number of characters that can be stored */

FILE *file_pointer; /* pointer to FILE data structure associated with
the file from which a line is read */

description The *fgets( )* function reads at most one less than the number of characters specified by *max_char* from the file pointed to by *file_pointer* into the buffer pointed to by *string*. No additional characters are read after the new-line character (which is retained). A null character is written immediately after the last character read into the buffer.

Example

```
FILE *fp;
char string[81];
if ((fp = fopen (“A:\\file1”, “r”)) == NULL)
    {
    printf (“fopen failed.\n”);
    exit (0);
    }
while (fgets (string, 80, fp) != NULL)
     printf (“%s\n”, string);
```

return If there are no errors, *fgets* returns the argument string. Otherwiae, it returns a NULL. Call *ferror* and *feof* to determine if there was an error or the file simply reached its end.

see also fgetc, fputc, fputs

fopen

purpose To open a file for buffered input and output operations.

syntax

```
FILE *fopen (const char *filename, const char *mode);
const char *filename;          /* name of file to be opened including
                                  drive and directory specification */
const char *mode;              /* type of access permitted */
```

description The *fopen()* function opens the file specified in the argument *filename*. The *filename* must include the drive, which is “**A:**”. If the operation fails, a null pointer is returned. The mode string specifies the type of access requested as follows:

“**r**” Open for reading in text mode

“**w**” Create for writing in text mode

“**a**” Append (open/create for writing at EOF)

“**rb**” Open for reading in binary mode

“**wb**” Create or truncate for writing in binary mode

“**ab**” Append in binary mode (open/create for writing at EOF)

“**r+**” Open for reading and writing in text mode

“**w+**” Truncate or create for reading and writing in text mode

“**a+**” Open / create for reading and appending.

“**r+b**” Open for reading and writing in binary mode

“**w+b**” Truncate or create for reading and writing in binary mode

“**a+b**” Open/create for reading and appending in binary mode

“**d**” Open directory

example

```
FILE *fp;
```

```
if ((fp = fopen ("A:\\file1", "r+")) == NULL)    /* opened for read & write */
    {
    printf ("fopen failed.\n");
    exit (0);
    }
```

return If the file is opened successfully, *fopen* returns a pointer to the file. Actually, this is a pointer to a structure of type FILE, which is defined in the header file **smc.h**. In case of an error, *fopen* returns a NULL. The value of the global *errno* may contain additional error status. See **smc.h** for the error codes returned in *errno*

see also fclose

fputc

purpose To write a single character to a file opened for buffered output.

syntax

```
int fputc (int c, FILE *file_pointer);
int  c;                    /* character to be written */
FILE *file_pointer;     /* pointer to the FILE data structure associated with
                           the file to which the character is to be written */
```

description The *fputc()* function writes a character given in the argument *c* to the file specified by the *file_pointer* in the current position and then increments this position after writing the character.

example

```
FILE *fp;
char buffer[81] = "Testing the function fputc";
int  i;
if ((fp = fopen ("A:\\file1", "w")) == NULL)
    {
    printf ("fopen failed.\n");
    exit (0);
    }
for (i=0; (i < 80) && (fputc (buffer[i], fp) != EOF); i++)
     ;
```

return If there are no errors, *fputc* returns the character written. Otherwise, it returns the constant EOF. Call *ferror* to determine if there was an error or the integer argument *c* just happened to be equal to EOF.

see also fgetc, fgets, fputs

fputs

purpose To write a null-terminated string to a file opened for buffered output.

syntax

```
int fputs (char *string, FILE *file_pointer);
char *string;            /* null-terminated character string to be output */
FILE *file_pointer;     /* pointer to the FILE data structure associated with
                           the file to which the string is output */
```

description The *fputs* function writes a string given in the argument *string* to the file specified by the *file_pointer*.

Example

```
FILE *fp;
char string[81] = “Testing the function fputs”;
if ((fp = fopen (“A:\\file1”, “w”)) == NULL)
    {
    printf (“fopen failed.\n”);
    exit (0);
    }
fputs (string, fp);
```

return If there are no errors, *fputs()* returns the number of characters written. Otherwise, it returns the constant EOF. Call *ferror()* to find out the error.

see also fgetc, fgets, fputc

fread

purpose To read a specified number of data items, each of a given size, from the current position in a file opened for buffered input.

syntax

```
int fread (void *buffer, int size, int count, FILE *file_pointer);
void *buffer;       /* pointer to memory where fread stores the bytes it
                       reads */
int size;           /* size in bytes of each data item */
int count;          /* maximum number of items to be read */
FILE *file_pointer; /* pointer to FILE data structure associated with
                       the file from which data items are read */
```

description The *fread()* function reads *count* data items, each of *size* bytes, starting at the current read position of the file specified by *file_pointer*. After the read is complete, the current position is updated.

Example

```
FILE *fp;
char buffer[81];
int  count;
if ((fp = fopen (“A:\\file1”, “r”)) == NULL)
    {
    printf (“fopen failed.\n”);
    exit (0);
    }
count = fread (buffer, 1, 80, fp);
printf (“Read these %d characters:\n %s\n”, count, buffer);
```

return The actual number of items read is returned. Note that the number of items returned will be equal to *count* unless the EOF is reached or some error occurs.

see also fwrite

fseek

purpose To reposition a file pointer.

syntax

```
int fseek (FILE *file_pointer, long offset, int origin);
```

FILE *file_pointer; /* pointer to FILE data structure associated with
the file whose position is to be set */
long offset; /* offset of new position (in bytes) from origin */
int origin; /* file position from which to add offset, there are three
values available :
SEEK_SET (1) - Beginning of file
SEEK_CUR (0) - Current file pointer position
SEEK_END (-1) - End of file */

description The *fseek( )* function repositions the file specified by *file_pointer* by *offset* bytes from *origin*. If the file is opened in text mode, the *offset* should be 0 or the value returned by *ftell( )*. The value in *origin* should be SEEK_SET for beginning of file, SEEK_CUR for current file pointer position, or SEEK_END for end of file.

Example

```
FILE *fp;
if (fseek(fp, 30L, SEEK_SET) != 0)
    printf ("fseek failed!");
```

return If successful, *fseek* returns a zero, otherwise, it returns a nonzero value.

see also ftell

fsetpos

purpose To set the position where reading or writing can take place in a file opened for buffered I/O.

syntax

```
int fsetpos (FILE *file_pointer, const unsigned long *pos);
FILE *file_pointer; /* pointer to FILE data structure associated with the
file whose position is to be set */
const unsigned long * pos;  /* pointer to location containing new value of
file position */
```

description The *fsetpos( )* function sets the file pointer associated with opened file to the new position *pos*. The new position is the value obtained by a previous call to *fgetpos( )* on that stream. The reason for the existence of *fgetpos* and *fsetpos* (in addition to *fseek*) is that if you want to position to a file in text mode, you cannot necessarily find a position by counting the characters you have written out, since text mode translation may change that number. In this case you can only use *fgetpos* to find a current position and then return there later with *fsetpos*.

Example

```
FILE *fp;
unsigned long curpos;
char  buffer [80];
if (fgetpos (fp, &curpos) != 0)          /* save current position */
    printf ("fgetpos failed!");
if (fgets(buffer, 20, fp) == NULL)       /* read 20 characters */
    printf ("fgets failed!");
if (fsetpos (fp, &curpos) != 0)          /* reset to previous position */
    printf ("fsetpos failed!");
```

return If successful, *fsetpos* returns a zero, otherwise, it returns a nonzero value with the global variable *errno* set to a nonzero error code.

see also fgetpos

ftell

purpose To get current file position.

syntax
```
long ftell (FILE *file_pointer);
FILE *file_pointer; /* pointer to FILE data structure associated with
                       the file whose current position is to be returned */
```

description The *ftell( )* function returns the current read and write position of the file specified by argument *file_pointer*.

Example
```
FILE *fp;
long curpos;
if ((curpos = ftell (fp)) == -1L)
    printf ("ftell failed!");
```

return If successful, *ftell* returns a long integer containing the number of bytes the current position is offset from the beginning of the file. In case of error, *ftell* returns –1L with the global variable *errno* set to a positive error code.

see also fseek

fwrite

purpose To write a specified number of data items, each of a given size, from a buffer to the current position in a file opened for buffered output.

syntax
```
int fwrite (const void *buffer, int size, int count, FILE *file_pointer);
const void *buffer;  /* pointer to buffer from which fwrite will get the
                        bytes it writes */
int size;            /* size in bytes of each data item */
int count;           /* maximum number of items to be written */
FILE *file_pointer;  /* pointer to FILE data structure associated with the
                        file from to which data items are to be written */
```

description The *fwrite()* function writes *count* data items, each of *size* bytes, to the file specified by the argument *file_pointer*, starting at the current position. After the write operation is complete, the current position is updated.

Example
```
FILE *fp;
char buffer[81] = "Testing the fwrite function";
int  count;
if ((fp = fopen ("A:\\file1", "r")) == NULL)
    {
    printf ("fopen failed.\n");
    exit (0);
    }
count = fwrite (buffer, 1, 20, fp);
```

printf (“%d characters written to a file”, count);

return The actual number of items written is returned. Note that the number of items returned will be equal to *count* except an error occurred.

see also fread

fremove

purpose To delete a file.

syntax int fremove (const char *filename);

const char *filename; /* the complete pathname of the file to delete */

description The *fremove( )* function deletes a file specified by *filename*. The complete filename should include the device name.

Example if (fremove (“a:\\subdir\\thisfile.txt”))

printf (“errno = %d\n”, errno);

return If successful, *fremove* returns a zero, otherwise, it returns a nonzero value with the global variable *errno* set to a nonzero error code (refer to *smc.h*).

see also frename, rmdir

frename

purpose To rename (or move) a file or a subdirectory.

syntax int frename (const char *oldname, const char *newname);

const char *oldname; /* the complete pathname of an existing file */

const char *newname; /* the complete pathname of the target file */

description The *frename( )* function changes the name of the file *oldname* to *newname*. A complete pathname must be given for both, which must be on the same device (drive). Subdirectories can also be renamed. The *newname* does not need to be in the same directory as *oldname*. The effect in this case is that of moving the file to the new directory (and possibly renaming it during the process).

Example if (frename(“a:\\file1.txt”, ”a:\\file2.txt”))

printf (“errno = %d\n”, errno);

return If successful, *frename* returns a zero, otherwise, it returns a nonzero value with the global variable *errno* set to a nonzero error code (refer to *smc.h*).

see also fremove, rmdir

mkdir

purpose To create a new directory.

syntax int mkdir (const char * path);

const char *path; /* the complete pathname of the directory to create */

description The *mkdir( )* function creates a new directory from the given pathname *path*.

Example if (mkdir (“A:\\thisdir\\thatdir\\newdir”) != 0)

printf (“Fail to create a directory”);

return If successful, *mkdir* returns a zero, otherwise, it returns a nonzero value with the global variable *errno* set to a nonzero error code (refer to *smc.h*).

see also rmdir

rmdir

purpose To remove (delete) a directory.

syntax int rmdir (const char * path);

const char *path; /* the complete pathname of the directory to delete */

description The *rmdir( )* function removes the directory specified by the argument *path* from the file system. The directory must be empty or an error is returned. An attempt to remove the root directory also returns an error.

Example

```
if (rmdir ("a:\\thisdir\\thatdir") != 0)
    printf ("Fail to delete the directory");
```

return If successful, *rmdir* returns a zero, otherwise, it returns a nonzero value with the global variable *errno* set to a nonzero error code (refer to *smc.h*).

see also mkdir

chmod

purpose To change the attributes of the given pathname file

syntax int chmod (const char * pathname, int attribute);

const char *pathname; /* the complete pathname to the file */

int attribute; /* new attribute value for the file */

description The *chmod( )* function will change the *attribute* associated with the file specified by *pathname*. The attributes must be one or more of the following:

FA_NORMAL	Normal file (no attributes)
FA_RDONLY	Read-only file
FA_HIDDEN	Hidden file (does not affect accessibility)
FA_SYSTEM	System file
FA_ARCH	Archive bit (file changed since bit cleared)

Example

```
int  att;
att = chmod ("a:\\myfile.bin", FA_SYSTEM | FA_RDONLY)
if (att == EOF)
    printf ("Chmod error, a:\\myfile.bin\n");
```

return If successful, *chmod* returns the new attributes, otherwise, it returns the constant EOF.

see also chmodfp

chmodfp

purpose To changes the attributes of a file by using pointer

syntax int chmodfp (FILE * file_pointer, int function, int attribute);

FILE *file_pointer; /* pointer to FILE data structure associated with
the file whose attribute is to be changed */

int function; /* 0 = return current, 1 = set new attribute */

int attribute; /* new attribute value for the file */

description The *chmodfp( )* function will either return, or change the attributes of the opened file specified by *file_pointer*. If *function* = 0, then the current file attributes are returned. If *function* = 1, then the file attributes are set to new *attribute*. The FA_DIR attribute cannot be changed by this function. The new attributes will have no effect until the file is closed and reopened (e.g., if the file is currently open for writing, and is made read-only by this function, writes to the file are still permitted until the file is closed and reopened). The attributes must be one or more of the following:

FA_NORMAL	Normal file (no attributes)
FA_RDONLY	Read-only file
FA_HIDDEN	Hidden file (does not affect accessibility)
FA_SYSTEM	System file
FA_ARCH	Archive bit (file changed since bit cleared)
FA_DIR	File is a subdirectory

Example

```
FILE *fp;
Int  att;
fp = fopen (“A:\\MYFILE.BIN”, “r+b”)
att = chmodfp (fp, 1, FA_SYSTEM | FA_RDONLY);
```

return If successful, *chmodfp* returns the current attributes of the file, otherwise, it returns the constant EOF.

see also chmod

chvlabel

purpose To changes an existing volume label

syntax

```
int chvlabel(const char *drivename, char *oldlabel, const char *newlabel);
const char *drivename;  /* name of drive to alter label on (e.g. “A:”) */
char *oldlabel;         /* pointer to where to return old label */
const char *newlabel;   /* the new label string to set */
```

description The *chvlabel( )* function returns the existing volume label of the specified drive in *oldlabel*. If no volume label currently exists, *oldlabel* will be set to an empty string. If *newlabel* does not equal NULL, then the *newlabel* string is made the current volume label.

Example

```
char old_label [12];
if (chvlabel (“A:”,old_label ,NULL))
   printf (“chvlabel failed”);
```

return If successful, *chmodfp* returns a zero, otherwise, it returns a nonzero value.

2.15 Miscellaneous

DownLoadPage

purpose Enter the 'Download' mode

syntax void DownLoadPage();

example call
```
open_com (1, 0x08);          /* 38400, N, 8 */
DownLoadPage();              /* enter download mode */
```

description The *DownLoadPage* function is used to set 711/720 to the download mode. The Download page will show up and user can select the communication port and the baud rate for program download.

return none

prc_menu

purpose Create a menu-driven interface.

syntax void prc_menu (MENU *menu);

example call
```
MENU MyMenu = {3, 1, 0, "My menu", {&Collect, &Upload, &Download}};
MENU_ENTRY Collect = {0, 1, "1. Collect", FuncCollect, 0};
MENU_ENTRY Upload = {0, 2, "2. Upload", FuncUpload, 0};
MENU_ENTRY Download = {0, 3, "3. Download", FuncDownload, 0};
Void FuncCollect (void)
    {
    /* to do: add your own program code here */
    }
Void FuncUpload (void)
    {
    /* to do: add your own program code here */
    }
Void FuncDownload (void)
    {
    /* to do: add your own program code here */
    }
prc_menu (&MyMenu);          /* process MyMenu menu*/
```

description The *prc_menu* function is used to create a user-defined menu. SMENU and MENU structures are defined in "711lib.h" and "720lib.h". Users can just fill the MENU structure and call the *prc_menu* function to build a hierarchy menu-driven user interface.

return none

3 Standard Library Routines

The standard library routines supported are categorized and listed below,

3.1 Input and Output : <stdio.h>

- File Operations: Not supported, please use Syntech Library routines.
- Formatted Output: Only sprintf is supported, for formatted output to display, please refer to Syntech Library "LCD".
- Formatted Input: Only sscanf is supported.
- Character Input and Output: Not supported, please refer to Syntech Library "External AT Keyboard" and "Membrane Keypad"
- Direct Input and Output: Not supported.

3.2 Character Class Test : <ctype.h>

For each function, the argument is a char, whose value must be EOF or representable as an unsigned char, and the return value is an int. The functions return non-zero (true) if the argument c satisfies the condition described, and zero if not.

- isalnum(c) isalpha(c) or isdigit(c) is true
- isalpha(c) isupper(c) or islower(c) is true
- iscntrl(c) control character
- isdigit(c) decimal digit
- isgraph(c) printing charcater except space
- islower(c) lower-case letter
- isprint(c) printing character including space
- ispunct(c) printing character except space or letter or digit
- isspace(c) space, formfeed, newline, carriage return, tab, vertical tab
- isupper(c) upper-case letter
- isxdigit(c) hexadecimal digit

In addition, there are two functions that convert the case of letters,

- int tolower(c) convert c to lower case
- int toupper(c) convert c to upper case

3.3 String Functions : <string.h>

Functions start with "str"

In the routine list, the type of variables used are as below,

```
char *s, t;
const char * cs, ct;
size_t n;
int c;
```

- char *strcpy(s, ct) copy string ct to string s, including 0x00, return s
- char *strncpy(s, ct, n) copy at most n characters of string ct to s, return s, pad with 0x00s if ct has fewer than n characters
- char *strcat(s, ct) concatenate string ct to end of string s, return s
- char *strncat(s, ct, n) concatenate at most n characters of ct to s, return s
- int strcmp(cs, ct) compare string cs and ct, return value < 0 if cs<ct, = 0 if cs = ct, > 0 if cs>ct
- int strncmp(cs, ct, n) compare at most n characters of string cs and ct, return value < 0 if cs < ct, = 0 if cs = ct, > 0 if cs>ct

- char *strchr(cs, c) return pointer to first occurrence of c in cs or NULL if not present
- char *strrchr(cs, c) return pointer to last occurrence of c in cs or NULL if not present
- size_t strspn(cs, ct) return length of prefix of cs consisting of characters in ct
- size_t strcspn(cs, ct) return length of prefix of cs consisting of characters not in ct
- char *strpbrk(cs, ct) return pointer to first occurrence in string cs of any character of string ct, or NULL if none are present
- char *strstr(cs, ct) return pointer to first occurrence of string ct in cs, or NULL if not present
- size_t strlen(cs) return length of string cs
- char *strtok(s, ct) searches s for tokens delimited by characters from ct
- strcoll Not supported
- strerror Not supported

Functions start with "mem"

In the list, types of variables are as below,

```
void *s, *t;
const void *cs, *ct;
size_t n;
int c;
```

- void *memcpy(s, ct, n) copy n characters from ct to s, return s
- void *memmove(s, ct, n) same as memcpy except that it works fine even if the objects overlap
- int memcmp(cs, ct, n) compare the first n characters of cs with ct; return as strcmp
- void *memchr(cs, c, n) return pointer to first occurrence of character c in cs or NULL if not present among the first n characters
- void *memset(s, c, n) place character c into first n characters of s, return s

3.4 Mathematical Functions : <math.h>

Mathematical functions are listed below and all of them return a double.

In the list, types of variables are as below,

```
double x, y;
int n;
```

- sin(x) sine of x
- cos(x) cosine of x
- tan(x) tangent of x
- asin(x) $\sin^{-1}(x)$ in range $[-\pi/2, \pi/2]$, $x \in [-1, 1]$
- acos(x) $\cos^{-1}(x)$ in range $[0, \pi]$, $x \in [-1, 1]$
- atan(x) $\tan^{-1}(x)$ in range $[-\pi/2, \pi/2]$
- atan2(y, x) $\tan^{-1}(y/x)$ in range $[-\pi, \pi]$
- sinh(x) hyprebolic sine of x
- cosh(x) hyperbolic cosine of x
- tanh(x) hyperbolic tangent of x
- exp(x) exponential function e^x
- log(x) natural logarithm ln(x), x>0
- log10(x) base 10 logarithm $\log_{10}(x)$, x>0
- pow(x, y) $_x^y$· A domain error occurs if x=0 and y<=0, or if x<0 and y is not an integer
- sqrt(x) x, x0

- ceil(x) smallest integer not less than x, as a double
- floor(x) largest integer not greater than x, as a double
- fabs(x) absolute value x
- ldexp(x, n) $x * 2^n$
- frexp(x, int *exp) splits x into a normalized fraction in the interval [1/2, 1], which is returned, and a power of 2, which is stored in *exp. If x is zero, both parts of the result are zero.
- modf(x, double *ip) splits x into integral and fractional parts, each with the same sign as x. It stores the integral part in *ip, and returns the fractional part.
- fmod(x, y) floating point remainder of x/y, with the same sign as x. If y is 0, the result is implementation-defined.

3.5 Utility Function : <stdlib.h>

Number Conversion

- double atof(const char *s) convert s to double, equivalent to strtod(s, (char **)NULL)
- int atoi(const char *s) convert s to integer, equivalent to strtol(s, (char**)NULL, 10)
- int atol(const char *s) convert s to long, equivalent to strtol(s, (char**)NULL, 10)
- double strtod(const char *s, char **endp) converts the prefix of s to double
- long strtol(const char *s, char **endp, int base) converts the prefix of s to long
- unsigned long strtoul(const char *s, char **endp, int base) converts the prefix of s to unsigned long
- int rand(void) returns a random integer from 0 to 32767
- void srand(unsigned int seed) seed for new pseudo-random generation
- void *bsearch() binary search
- void qsort() ascending sorts
- int abs(int n) integer absolute
- long labs(long n) long absolute
- div_t div(int num, int denom) integer division
- ldiv_t ldiv(long num, long denom) long division

Storage Allocation

Not supported. Please use Syntech library routines instead.

3.6 Diagnostics : <assert.h>

Not supported.

3.7 Variable Argument Lists : <stdarg.h>

Functions for processing variable arguments are listed below.

va_start(va_list ap, lastarg)
type va_arg(va_list ap, type)
void va_end(va_list ap)

3.8 Non-Local Jumps : <setjmp.h>

Not supported.

3.9 Signals : <signal.h>

Not supported.

3.10 Date and Time Function : <time.h>

Not supported.

3.11 Implementation-defined Limits : <limits.h> and <float.h>

Please refer to limit.h and float.h.

4 Real Time Kernel

All Portable Terminals come with a real-time kernel (μC/OS) that allows the user to generate a preemptive multitasking application. The user can apply the real time kernel functions to split the application into multiple tasks that each task takes turn to gain the access to the system resource by a priority-based schedule.

μC/OS applies the semaphore mechanism to control the access to the shared resource for the multiple tasks. There are generally only three operations that can be performed on a semaphore: CREATE, PEND, and POST. A semaphore is a key that the task requires in order to continue execution. If the semaphore is already in use, the requesting task is suspended until the semaphore is released by its current owner.

A task is an infinite loop function or a function which deletes itself when it is done executing. Each task is assigned with an appropriate priority. The more important the task, the higher the priority given to it. μC/OS can manage up to 32 tasks (with priority 0 to 31, the lower number, the higher priority) for the user's program of the 711/720 Data Terminal. The main task, main( ), takes priority 16.

A task desiring the semaphore will perform a PEND operation. A task releases a semaphore by performing a POST operation. If there are several tasks on the pending list, the highest priority task waiting for the semaphore will receive the semaphore when the semaphore is posted. The pending list of tasks is always initially empty.

The μC/OS related functions are discussed as follows.

OS_ENTER_CRITICAL

purpose Disable the processor's interrupt

syntax void OS_ENTER_CRITICAL(void);

example call
```
OS_ENTER_CRITICAL();
        … /* user code */
OS_EXIT_CRITICAL();
```

description A critical section of code is code that needs to be treated indivisibly. Once the section of code starts executing, it must not be interrupted. To ensure this, user can call *OS_ENTER_CRITICAL* function to disable interrupts prior to executing the critical code and enable the interrupts when the critical code is done. The function executes in about 5 CPU clock cycles. This function and *OS_EXIT_CRITICAL* function must be used in pairs.

return none

OS_EXIT_CRITICAL

purpose Ensable the processor's interrupt

syntax void OS_EXIT_CRITICAL(void);

example call
```
OS_ENTER_CRITICAL();
        … /* user code */
OS_EXIT_CRITICAL();
```

description The function executes in about 5 CPU clock cycles. This function and *OS_ENTER_CRITICAL* function must be used in pairs.

return none

OSSemCreate

purpose Create and initialize a semaphore

syntax OS_EVENT OSSemCreate(unsigned *value*);

where, OS_EVENT, a data structure to maintain the state of an event called Event Control Block (ECB), is defined as below,

```
typedef struct os_event {
    unsigned char  OSEventTbl[8];  /* Group corresponding to tasks
                                      waiting for event to occur */
    unsigned char  OSEventGrp;     /* List of tasks waiting for event to
                                      occur */
    long  OSEventCnt;              /* Count of used when event is a
                                      semaphore */
    void  *OSEventPtr;             /* Pointer to message or queue
                                      structure */
} OS_EVENT;
```

value is the initial value of the semaphore. The initial *value* of the semaphore is allowed to be between 0 and 32767.

example call sem_time = OSSemCreate(1); /* create a semaphore sem_time and the initial value of sem_time is set to 1. */

description This function is used to create and initialize a semaphore. Semaphores must be created before they are used.

return A pointer to the event control block allocated to the semaphore. If no event control block is available, a NULL pointer will be returned.

OS_NO_ERR,if the function was successful.

OSSemPend

purpose List a task on the pending list for the semaphore

syntax unsigned char OSSemPend (OS_EVENT **pevent*, unsigned long *timeout*, unsigned char **err*);

where, *pevent* is a pointer to the semaphore. This pointer is returned to your application when the semaphore is created.

timeout is used to allow the task to resume execution if the semaphore is not acquired within the specified number of clock ticks. A *timeout* value of 0 indicates that the task desires to wait forever for the semaphore. The maximum *timeout* is 65535 clock ticks.

err is a pointer to a variable which will be used to hold an error code. *OSSemPend* sets *err to either:

(1) OS_NO_ERR, if the semaphore is available
(2) OS_TIMEOUT, if a timeout occurred

example call OSSemPend (sem_time, 0, &err);

description This function is used when a task desires to get exclusive access to a resource, synchronize its activities with an Interrupt Service Routine (ISR) or wait until an event occurs. If a task calls *OSSemPend* function and the value of the semaphore is greater than 0, then *OSSemPend* function will decrement the semaphore and return to its caller. However, if the value of the semaphore is less than or equal to zero, *OSSemPend* function decrements the semaphore value and places the calling task in the waiting list for the semaphore. The task will thus wait until a task or an ISR releases the semaphore or signals the occurrence of the event. In this case, rescheduling occurs and the next highest priority task ready to run is given control of the CPU. An optional timeout may be specified when pending for a semaphore.

return none

OSSemPost

purpose Signal the semaphore

syntax unsigned char OSSemPost (OS_EVENT **pevent*);

where, *pevent* is a pointer to the semaphore. This pointer is returned to your application when the semaphore is created.

example call OSSemPost (sem_time);

description A semaphore is signaled by calling *OSSemPost* function. If the semaphore value is greater than or equal to zero, the semaphore is incremented and *OSSemPost* function returns to its caller. If the semaphore value is negative then tasks are waiting for the semaphore to be signaled. In this case, *OSSemPost* function removes the highest priority task pending for the semaphore from the waiting list and makes this task ready to run. The schedule is then called to determine if the awakened task is now the highest priority task ready to run

return (1) OS_NO_ERR, if the semaphore is available

(2) OS_TIMEOUT, if a timeout occurred

OSTaskCreate

purpose Create a task

syntax unsigned char OSTaskCreate (void (**task*)(void *pd), void **pdata*, unsigned char **pstk*, unsigned long stk_size, unsigned char *piro*);

where, *task* is a pointer to the task's code.

pdata is a pointer to an optional data area which can be used to pass parameters to the task when it is created.

pstk is a pointer to the task's top of stack. The stack is used to store local variables, function parameters and return addresses and CPU registers during an interrupt. The size of this stack is defined by the task requirements and the anticipated interrupt nesting. Determining the size of the stack involves knowing how many bytes are required for storage of local variables for the task itself, all nested functions, as well as requirements for interrupts (accounting for nesting).

prio is the task priority. A unique priority number must be assigned to each task and the lower the number, the higher the priority.

example call OSTaskCreate (beep_task, (void *)0, beep_stk, 256, 10); /* create a beep_task with priority 10 */
static unsigned char beep_stk[256];
void beep_task(void*);

description This function allows an application to create atask. The task is managed by μC/OS. Tasks can be created prior to the start of multitasking or by a running task.

return OS_PRIO_EXIST,if the requested priority already exist.
OS_NO_ERR,if the function was successful.

OSTaskDel

purpose To delete a task

syntax unsigned char OSTaskDel (unsigned char *piro*);
where, *prio* is the task priority. A unique priority number must be assigned to each task and the lower the number, the higher the priority.

example call OSTaskDel (10); /* delete a task with priority number 10 */

description This function allows user's application to delete a task by specifying the priority number of the task to delete. The calling task can be deleted by specifying its own priority number. The deleted task is returned to the dormant state. The deleted task may be created to make the deleted task active again.

return

OS_TASK_DEL_IDLE	if the task to delete is an idle task.
OS_TASK_DEL_ERR	if the task to delete does not exist.
OS_NO_ERR	if the task was deleted.

OSTimeDly

purpose	Allow a task to delay itself for a number of clock ticks.
syntax	void OSTimeDly (unsigned long *ticks*); where, *ticks* is the delay time in units of 5 ms.
example call	OSTimeDly(10); /* delay the task for 10 X 5 ms */
description	This function allows a task to delay itself for a number of clock ticks. Rescheduling always occurs when the number of clock ticks is greater than zero. Valid delays range from 1 to 65535 ticks. Note that calling this function with a delay of 0 results in no delay and thus the function returns to the caller.
return	none